Abstract Machines for Open Call-by-Value

Beniamino Accattoli, Giulio Guerrieri

Science of Computer Programming 184 (2019) 102275

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Abstract machines for Open Call-by-Value ☆,☆☆

Beniamino Accattoli [a], Giulio Guerrieri [b,*]

[a] INRIA, UMR 7161, LIX, École Polytechnique, Palaiseau, France
[b] University of Bath, Department of Computer Science, Bath, United Kingdom

ARTICLE INFO

Article history:
Received 17 February 2018
Received in revised form 17 January 2019
Accepted 8 March 2019
Available online 15 March 2019

Keywords:
Lambda-calculus
Cost model
Complexity
Size explosion
Implementation

ABSTRACT

The theory of the call-by-value λ-calculus relies on weak evaluation and closed terms, that are natural hypotheses in the study of programming languages. To model proof assistants, however, strong evaluation and open terms are required. *Open call-by-value* is the intermediate setting of weak evaluation with (possibly) open terms, on top of which Grégoire and Leroy designed one of the abstract machines of Coq. This paper provides a theory of abstract machines for the *fireball calculus*, the simplest presentation of open call-by-value.

The literature contains machines that are either *simple* but inefficient, as they have an exponential overhead, or *efficient* but heavy, as they rely on a labeling of environments and a technical optimization. We introduce a machine that is *simple* and *efficient*: it does not use labels and it implements the fireball calculus within a bilinear overhead. Moreover, we provide a new fine understanding of how different optimizations impact on the complexity of the overhead, and evidence that the time cost model we work with is minimal.

1. Introduction

The λ-calculus is the computational model behind functional programming languages and proof assistants. Its elegant definition is based on just one *import-step* computational rule, *β-reduction*, and does not rest on any notion of machine or automaton. Compilers and proof assistants however are concrete tools that implement the λ-calculus in some way—a problem clearly arises. There is a huge gap between the abstract mathematical setting of the calculus and the technical intricacies of an actual implementation. This is why the issue of implementation is studied via intermediate *abstract machines*, that are implementation schemes with *micro-step* operations and without too many concrete details.

Closed and strong λ-calculus. Functional programming languages are based on a simplified form of λ-calculus, that we like to call *closed λ-calculus*, with two important restrictions. First, evaluation is *weak*, *i.e.* it does not evaluate function bodies (until their formal parameters are replaced by actual arguments, if any). Second, terms are *closed*, that is, they have no free variables. The theory of the closed λ-calculus is much simpler than the general one (*e.g.*, see pp. 2–3).

Proof assistants based on the λ-calculus usually require the power of the full theory. Evaluation is then *strong*, *i.e.* without the two above restrictions, and the distinction between open and closed terms no longer makes sense, because evaluation

☆ This is a revised and extended version of [8].
☆☆ This work is part of a wider research effort, the COCA HOLA project (https://sites.google.com/site/beniaminoaccattoli/coca-hola).
* Corresponding author.
E-mail addresses: beniamino.accattoli@inria.fr (B. Accattoli), g.guerrieri@bath.ac.uk (G. Guerrieri).

https://doi.org/10.1016/j.scico.2019.03.002
0167-6423/

has to deal with the issues of open terms even if terms are closed, when it enters function bodies. We refer to this setting as the *strong λ-calculus*.

Historically, the study of strong and closed λ-calculi have followed orthogonal approaches. Theoretical studies rather dealt with the strong λ-calculus, and it is only since the seminal work of Abramsky and Ong [1] that theoreticians started to take the closed case seriously. Dually, practical studies mostly ignored strong evaluation, with the notable exception of Crégut [19] and some recent work [25,13,2]. Strong evaluation is however essential in the implementation of proof assistants or higher-order logic programming, typically for type-checking with dependent types as in the Edinburgh Logical Framework [28] or the Calculus of Constructions [18], and for unification in simply typed frameworks like λ-Prolog [29].

Open call-by-value. In recent work [7], we advocated the relevance of the *open λ-calculus*, a framework in between the closed and the strong ones, where evaluation is *weak* but terms may be *open*. Its key property is that the strong case can be described as the iteration of the open one into function bodies. The same cannot be done with the closed λ-calculus because—as already pointed out—entering function bodies requires to deal with (locally) open terms.

The open λ-calculus did not emerge before because most theoretical studies focus on the *call-by-name* strong λ-calculus, and in call-by-name the open/closed distinction does not play an important role. Such a distinction, instead, is delicate for *call-by-value* (CbV for short) evaluation,[1] where Plotkin's original operational semantics [31] is not adequate for open terms. We discussed this issue at length in [7], where four extensions of Plotkin's semantics to (possibly) open terms are compared and shown to be equivalent. That paper then introduces the expression *Open Call-by-Value* (shortened *Open CbV*) to refer to them as a whole, as well as *Closed CbV* and *Strong CbV* to concisely refer to the closed and strong CbV λ-calculi.

The fireball calculus. The simplest presentation of Open CbV (see Proposition 1-2 in Sect. 2) is the *fireball calculus* λ_{fire}, obtained from the CbV λ-calculus by generalizing values into *fireballs*. Dynamically, β-redexes can fire only when the argument is a fireball (*fireball* is a pun on *fire-able*). Fireballs extend values by adding *inert terms*, which are terms of (recursive) form $xf_1 \dots f_n$, where the f_i's are fireballs. Inert terms are always open, and so fireballs are simply abstractions in Closed CbV—notably, then, on closed terms λ_{fire} *coincides* with Plotkin's (Closed) CbV λ-calculus.

The fireball calculus was introduced without a name by Paolini and Ronchi Della Rocca [30,32], then rediscovered independently first by Grégoire and Leroy [27], and then by Accattoli and Sacerdoti Coen [10], who also named it.

Coq by (open) levels. In 2002, Grégoire and Leroy used the fireball calculus to improve the implementation of the Coq proof assistant [27]. They implemented Strong CbV by factoring such a task through Open CbV. They design an abstract machine for the fireball calculus—that in our paper is called Open GLAM (see Sect. 4 for details)—and then iterate it to evaluate Strong CbV *by (open) levels*: the Open GLAM is first executed at top level (that is, out of all abstractions), and then re-launched recursively under abstractions. Their study is itself formalized in Coq, but it lacks an estimation of the efficiency of the machine. It turns out that the Open GLAM is inefficient, to the point of being *unreasonable*.

In order to continue our story some basic facts about cost models and abstract machines have to be recalled (see [3] for a gentle tutorial, and [4] for a more general perspective about reasonable cost models for the λ-calculus).

Interlude 1: size explosion. In λ-calculi, the number of β-steps is the natural candidate as a time cost model. However, it is well-known that λ-calculi suffer from a degeneracy called *size explosion*: there are families of terms whose size is linear in n, that evaluate in n β-steps, and whose result has size exponential in n. So, the number of β-steps does not seem to be a reasonable cost model, because it does not even account for the time to write down the result of a computation—the *macro-step* character of β-reduction seems to forbid to count 1 for each β-step. This problem affects all λ-calculi and all evaluation strategies.

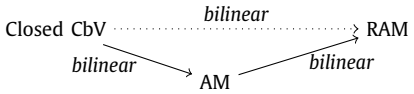
Interlude 2: reasonable cost models and abstract machines. Despite size explosion, surprisingly, for many strategies the number of β-steps *is* a reasonable time cost model, so that each single β-step can be seen as an elementary computation step and counted as 1 in such a model. There is no contradiction: λ-calculi can be simulated in alternative formalisms employing some form of *sharing*, such as abstract machines. These settings manage a compact representation of terms (in particular, of the result of evaluation) via *micro-step* operations, avoiding size explosion. Showing that a certain λ-calculus is reasonable is usually done by simulating it with a *reasonable* abstract machine, *i.e.* a machine implementable on a random access machine with asymptotic overhead *polynomial* in *two* parameters: the number of β-steps in the calculus and the size of the initial term. If the overhead is *bilinear* (*i.e.* linear in both parameters), the machine and the calculus it implements are *efficient*.

The design of a reasonable abstract machine depends very much on the kind of λ-calculus to be implemented, as different calculi admit different forms of size explosion and/or require more sophisticated forms of sharing. For strategies in the closed λ-calculus it is enough to use the ordinary technology for abstract machines, as first shown by Blelloch and

[1] In CbV evaluation, a function's arguments are evaluated before being passed to the function, so that β-redexes can fire only when their arguments are values, *i.e.* abstractions or variables. The idea of CbV evaluation is that only values can be duplicated or discarded.

Greiner [17] (1995), and then by Sands, Gustavsson, and Moran [33] (2002), and, with other techniques, by combining the results of Dal Lago and Martini [20] and [21] (2009).

Let us point out some details. Ordinary abstract machines (AM) for Closed CbV provide reasonable implementations (up to sharing) on random access machines (RAM) with a *bilinear* overhead (that is, linear in the number of β-steps *and* in the size of the initial term, so these implementations are actually efficient). The situation can be depicted as follows:



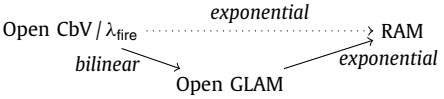
(1)

With an attentive choice of the data structures it is also possible to be *logarithmic* in the size of the initial term, as recently shown by Accattoli and Barras [5].

The case of the strong λ-calculus is subtler and a more sophisticated sharing is necessary, as shown by Accattoli and Dal Lago [6]. The topic of our paper is the study of reasonable abstract machines for the intermediate case of Open CbV.

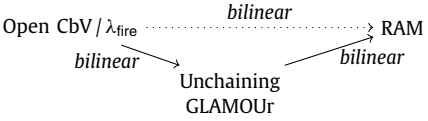
Fireball is reasonable. Accattoli and Sacerdoti Coen [10] studied Open CbV from the point of view of time cost models. Their work provides three contributions:

1. *Open size explosion*: they show that Open CbV is subtler than Closed CbV by exhibiting a form of size explosion that is not possible in Closed CbV, making Open CbV closer to Strong CbV rather than to Closed CbV. Essentially, this means that ordinary abstract machines for Open CbV do not provide reasonable implementations. For Grégoire and Leroy's Open GLAM the following diagram holds:



(2)

2. *Fireballs are reasonable*: they show that the number of β-steps in λ_{fire} is nonetheless a reasonable time cost model by exhibiting a refined abstract machine, called GLAMOUr, that they prove to be reasonable;
3. *And even efficient*: they also optimize the GLAMOUr into the Unchaining GLAMOUr, with a bilinear overhead, recasting diagram (1) in Open CbV:



(3)

This is an *optimal* solution. The speed-up shown by Accattoli and Barras [5] (lowering the dependence on the size of the initial term to logarithmic) is specific to the evaluation of closed terms and does not lift to open terms.

The fine art of implementing Open CbV. Here we introduce two new abstract machines, the Easy GLAMOUr (in Sect. 5–6) and the Fast GLAMOUr (in Sect. 7), that are proved to be correct implementations of Open CbV (more precisely, of the right-to-left evaluation strategy in λ_{fire}, defined in Sect. 2) with a polynomial and bilinear overhead, respectively. Our study refines the results of Accattoli and Sacerdoti Coen [10] along four axes:

1. *Simpler machines*: both the GLAMOUr and the Unchaining GLAMOUr of [10] are sophisticated machines relying on a labeling of terms. The unchaining optimization of the second machine is also quite heavy. Both the Easy GLAMOUr and the Fast GLAMOUr, instead, do not need labels and the Fast GLAMOUr is bilinear without the unchaining optimization.
2. *Simpler analyses*: the correctness and complexity analyses of the (Unchaining) GLAMOUr are developed in [10] thanks to an informative but complex decomposition via explicit substitutions, by means of the distillation methodology [12]. Here, instead, we decode the Easy GLAMOUr and the Fast GLAMOUr directly to the fireball calculus, that turns out to be much simpler. Moreover, the complexity analysis of the Fast GLAMOUr, surprisingly, turns out to be straightforward.
3. *Modular decomposition of the overhead*: we provide a fine analysis of how different optimizations impact on the complexity of the overhead of abstract machines for Open CbV, and how they can be modularly composed, with modular effects on the overhead. In particular, we show that an optimization considered essential in [10], namely *substituting abstractions on-demand*, is not mandatory for reasonable machines—the Easy GLAMOUr does not implement it and yet it is reasonable.
4. *Understanding Strong CbV*: we avoid—on purpose—the study of Strong CbV, and yet our study provides insights into the complexity of implementing it, independently of how it is defined. We show, indeed, that substituting abstractions on-demand can be avoided only as long as one stays *inside* Open CbV, whereas it is *mandatory* for Strong CbV. Substituting abstractions on-demand is an optimization introduced by Accattoli and Dal Lago [6] and currently no proof assistant implements it. Said differently, our work shows that the technology currently in use in proof assistants is, at least theoretically, unreasonable.

Table 1
The fireball calculus λ_{fire}.

Terms	$t, u, s, r ::= x \mid \lambda x.t \mid tu$
Fireballs	$f, f', f'' ::= \lambda x.t \mid i$
Inert Terms	$i, i', i'' ::= x \mid if$
Evaluation Contexts	$C ::= \langle\cdot\rangle \mid tC \mid Ct$
Rule at Top Level	Contextual closure
$(\lambda x.t)(\lambda y.u) \mapsto_{\beta_\lambda} t\{x \leftarrow \lambda y.u\}$	$C\langle t\rangle \to_{\beta_\lambda} C\langle u\rangle$ if $t \mapsto_{\beta_\lambda} u$
$(\lambda x.t)i \mapsto_{\beta_i} t\{x \leftarrow i\}$	$C\langle t\rangle \to_{\beta_i} C\langle u\rangle$ if $t \mapsto_{\beta_i} u$
Reduction	$\to_{\beta_f} := \to_{\beta_\lambda} \cup \to_{\beta_i}$

Summing up, our work does not improve the known bound on the asymptotic overhead of abstract machines for Open CbV, as the one obtained in [10] is already optimal. Our contributions instead are a simplification and a finer understanding of the subtleties of implementing Open CbV: we introduce *simpler* but still *reasonable* (and in some cases even *efficient*, as in [10]) abstract machines whose complexity analyses are elementary, and we carry a new modular view of how different optimizations impact on the (asymptotic) complexity of the overhead. In particular, while [10] shows that Open CbV is subtler than Closed CbV, here we show that Open CbV is simpler than Strong CbV, and that defining Strong CbV as iterated Open CbV, as done by Grégoire and Leroy [27], may introduce an explosion of the overhead, if done naively.

This journal paper is a revised and extended version of [8]. The additions with respect to [8] are:

- *Proofs*: detailed proofs of all claims (except for some ones already proved in the literature). Some proofs that are straightforward or trivially obtained from very similar ones in the body of this paper are moved to Appendix B.
- *Related machines*: definitions and explanations of the abstract machines closely related to the new ones, namely the simple but unreasonable one by Grégoire and Leroy [27], reformulated according to our conventions and named Open GLAM (in Sect. 4), and the reasonable but complex ones by Accattoli and Sacerdoti Coen [10], namely the GLAMOUr and the Unchaining GLAMOUr (in Sect. 9).
- *Examples and insights*: we provide more examples of machine executions together with refined explanations and insights. In particular, we stress the *commutation* between evaluation and the substitution of inert terms as the key abstract property leading to reasonable machines for Open CbV.
- *Minimality of the cost model*: formal evidence that the number of steps in the fireball calculus is a minimal time cost model (in Sect. 10). Technically speaking we do not prove minimality—that would require a proof of the non-existence of asymptotically faster implementations, and it is not even clear how one could prove it. Nonetheless, our rigorous examples show that a more parsimonious cost model would require some radically stronger implementation technology.

At the end of the paper, Appendix A contains a glossary of rewriting theory and the explanation of some notations.

2. The fireball calculus λ_{fire} & open size explosion

In this section we introduce the *fireball calculus* λ_{fire}, the presentation of Open CbV we work with in this paper, and show the example of size explosion particular to the open setting. We studied alternative presentations of Open CbV in [7,9].

The fireball calculus. The fireball calculus λ_{fire} is defined in Table 1. The idea is that the values of the CbV λ-calculus—*i.e.* abstractions $\lambda x.t$ and variables $x, y, z \ldots$—are generalized to *fireballs*, by extending variables to more general *inert terms*. Actually fireballs (noted $f, f', \ldots$) and inert terms (noted $i, i', \ldots$) are defined by mutual induction (in Table 1). For instance, $\lambda x.y$ is a fireball as an abstraction, while x, $y(\lambda x.x)$, xy, and $(z(\lambda x.x))(zz)(\lambda y.(zy))$ are fireballs as inert terms. All and only the terms of the form $xf_1 \ldots f_n$ (where $n \geq 0$ and all the f_i's are fireballs) are inert. The main feature of inert terms is that they are open (with a free "head variable"), normal (with respect to weak evaluation, see below), and when plugged in a context they cannot create a redex (in particular, they are not abstractions), hence the name.[2]

Application is left-associative, so tur stands for the term $(tu)r$. Terms are always identified up to α-equivalence and the set of free variables of a term t is denoted by $\mathtt{fv}(t)$: we say that t is *closed* if $\mathtt{fv}(t) = \emptyset$, otherwise t is *open*. We use $t\{x \leftarrow u\}$ for the term obtained by the capture-avoiding substitution of u for each free occurrence of the variable x in t.

To define evaluation in λ_{fire}, we use *evaluation contexts* (noted C), *i.e.* terms with exactly one occurrence of the *hole* $\langle\cdot\rangle$, an additional place-holder. We use $C\langle t\rangle$ for the term obtained by replacing the hole $\langle\cdot\rangle$ in the context C with the term t.

Evaluation is given by (non-deterministic) *call-by-fireball* β-reduction $\to_{\beta_f}$: the β-rule can fire, *lighting* the argument, only if the argument is a fireball (*fireball* is a catchier version of *fire-able* term). We actually distinguish two sub-rules: one

[2] In the literature there is a similar notion, *neutral term*, notably in Girard's version of reducibility candidates [26] where it denotes a term that is not an abstraction. However, the expression *neutral* often (*e.g.* see [13]) refers to terms that furthermore are (strongly) β-normal. Inert terms are yet another notion (non-abstractions that are weakly β-normal, that is, there can be β-redexes under abstractions), which is why we avoid calling them *neutral*. In Grégoire and Leroy [27], inert terms are called accumulators, and fireballs are simply called values.

that *lights* abstractions, noted $\mapsto_{\beta_\lambda}$, and one that *lights* inert terms, noted $\mapsto_{\beta_i}$ (see Table 1). Reductions $\to_{\beta_\lambda}$ and $\to_{\beta_i}$ are just the closure of the root-steps $\mapsto_{\beta_\lambda}$ and $\mapsto_{\beta_i}$, respectively, under evaluation contexts C. Reduction $\to_{\beta_f}$ is defined as the union of $\to_{\beta_\lambda}$ and $\to_{\beta_i}$, or equivalently, as the closure of the root-step $\mapsto_{\beta_f} = \mapsto_{\beta_\lambda} \cup \mapsto_{\beta_i}$ under evaluation contexts. Note that, according to the definition of evaluation contexts C, evaluation is *weak* (*i.e.* it does not reduce under abstractions).

Main properties of the calculus. A famous key property of Closed CbV (whose evaluation is $\to_{\beta_\lambda}$ restricted to closed terms) is *harmony*: given a *closed* term t, either it diverges or it evaluates to an abstraction, *i.e.* t is β_λ-normal if and only if t is an abstraction. The fireball calculus λ_{fire} satisfies an analogous property in the *open* setting by replacing abstractions with fireballs (Proposition 1.1). Moreover, the fireball calculus is a *conservative extension* of Closed CbV: on closed terms it collapses on Closed CbV (Proposition 1.2). No other presentation of Open CbV has these good properties, which together with its simplicity are the reason to adopt it as the best presentation of Open CbV in order to study its implementations.

Proposition 1 *(Distinctive properties of λ_{fire}). Let t be a term.*

1. Open harmony*: t is β_f-normal if and only if t is a fireball.*
2. Conservative open extension*: $t \to_{\beta_f} u$ if and only if $t \to_{\beta_\lambda} u$, for t closed.*

Proof. 1. ($\Rightarrow$) Proof by induction on t. If t is a variable or an abstraction then t is a fireball. Otherwise $t = us$ for some terms u and s; since t is β_f-normal, then u and s are β_f-normal, and u is not an abstraction or s is not a fireball; by *i.h.*, u and s are fireballs; summing up, u is an inert term (because it is a fireball that is not an abstraction) and s is a fireball, thus $t = us$ is an inert term and so a fireball.

($\Leftarrow$) By hypothesis, t is an abstraction or an inert term. If t is an abstraction, it is β_f-normal since $\to_{\beta_f}$ does not reduce under λ's. Otherwise t is inert, and we prove by induction on the definition of inert term that t is β_f-normal.

- If t is a variable, then t is obviously β_f-normal.
- If $t = i(\lambda x.u)$ then i is β_f-normal by *i.h.*, and $\lambda x.u$ is β_f-normal as we have just shown; also, i is not an abstraction, so t is β_f-normal.
- Finally, if $t = ii'$ then i and i' are β_f-normal by *i.h.*, moreover i is not an abstraction, hence t is β_f-normal.

2. ($\Rightarrow$) The idea of the proof is that inert terms are always open, thus closed fireballs are simply abstractions. Formally, the proof is by induction on the definition of $t \to_{\beta_f} u$. Cases:

- *Step at the root, i.e.* $t = (\lambda x.s)f \mapsto_{\beta_f} s\{x \leftarrow f\} = u$. Since t is closed, then f is closed and hence cannot be an inert term, so f is a (closed) abstraction and thus $t = (\lambda x.s)f \mapsto_{\beta_\lambda} s\{x \leftarrow f\} = u$.
- *Application left, i.e.* $t = sr \to_{\beta_f} s'r = u$ with $s \to_{\beta_f} s'$. Since t is closed, s is so and hence $s \to_{\beta_\lambda} s'$ by *i.h.*; therefore, $t = sr \to_{\beta_\lambda} s'r = u$.
- *Application right, i.e.* $t = rs \to_{\beta_f} rs' = u$ with $s \to_{\beta_f} s'$. Analogous to the previous case.

($\Leftarrow$) By definition, $\to_{\beta_\lambda} \subseteq \to_{\beta_f}$ (recall that an abstraction is a fireball). □

The rewriting rules of λ_{fire} have also many good operational properties that we studied in [7], summarized in the following claim (proved in [7, Proposition 3]).

Proposition 2 *(Operational properties of λ_{fire}, [7]). The reduction $\to_{\beta_f}$ is quasi-diamond. If there is a β_f-normalizing derivation from a term t, then t cannot β_f-diverge and all β_f-normalizing derivations d from t have the same length $|d|_{\beta_f}$, the same number $|d|_{\beta_\lambda}$ of β_λ-steps, and the same number $|d|_{\beta_i}$ of β_i-steps.*

Right-to-left evaluation. As expected from a *calculus*, evaluation $\to_{\beta_f}$ in λ_{fire} is *non-deterministic*, because in an application there is no fixed order to evaluate the left or right subterm. Abstract machines however implement *deterministic* strategies. We then fix a deterministic strategy (which fires β_f-redexes from right to left and is the one implemented by the machines of the next sections). By Proposition 2, the choice of the strategy does not impact either on existence of a result (any strategy normalizes, if there is a normal form), or on the result itself (uniqueness of the normal form) or on the number of steps to reach it. It does impact however on the design of the machine, which selects β_f-redexes from right to left.

The *right-to-left evaluation strategy* $\to_{\mathtt{r}\beta_f}$ is defined by closing the root-step $\mapsto_{\beta_f} = \mapsto_{\beta_\lambda} \cup \mapsto_{\beta_i}$ in Table 1 under *right contexts* R, a special kind of evaluation context defined by:

$$\textit{Right Contexts} \qquad R ::= \langle\cdot\rangle \mid tR \mid Rf$$

(so, $\to_{\mathtt{r}\beta_f} \subsetneq \to_{\beta_f}$). The next lemma ensures that our definition is correct. We say that $(\lambda x.u)f$ is a *β_f-redex* (resp. *$\mathtt{r}\beta_f$-redex*) *in* t if $t = C\langle(\lambda x.u)f\rangle$ (resp. $t = R\langle(\lambda x.u)f\rangle$) for some evaluation (resp. right) context C (resp. R). Clearly, t is β_f-normal (resp. $\mathtt{r}\beta_f$-normal) if and only if t has no β_f-redex (resp. $\mathtt{r}\beta_f$-redex).

Lemma 1 *(Properties of $\to_{\mathtt{r}\beta_f}$). Let t be a term.*

1. Completeness*: t has a β_f-redex if and only if t has an $\mathtt{r}\beta_f$-redex.*
2. Determinism*: t has at most one $\mathtt{r}\beta_f$-redex.*

Proof. 1. ($\Leftarrow$) Immediate, since $\to_{\mathtt{r}\beta_f} \subseteq \to_{\beta_f}$.

($\Rightarrow$) Let C be the evaluation context of the rightmost β_f-redex in t, *i.e.* $t = C\langle(\lambda x.r)\,f\rangle$ and if $t = C'\langle(\lambda x'.r')\,f'\rangle$ for some $C' \neq C$ then the hole in C is "more on the right" than in C'. We show that C is a right context, by induction on C. Cases:

(a) *Empty*, *i.e.* $C = \langle\cdot\rangle$. Then clearly C is a right context.
(b) *Application right*, *i.e.* $t = us$ and $C = uC'$. As the rightmost β_f-redex in t is in s, then C' is a right context by *i.h.*, and so C is a right context.
(c) *Application left*, *i.e.* $t = us$ and $C = C's$. As the rightmost β_f-redex in t is in u, then C' is a right context by *i.h.*, while s is β_f-normal and so a fireball by open harmony (Proposition 1.1). Thus, C is a right context.

2. By induction on t. By completeness of $\to_{\mathtt{r}\beta_f}$ (Point 1), open harmony (Proposition 1.1) holds for $\to_{\mathtt{r}\beta_f}$: a term is $\mathtt{r}\beta_f$-normal (*i.e.* has no $\mathtt{r}\beta_f$-redexes) if and only if it is a fireball. We use this fact implicitly in the following case analysis. If t is a variable or an abstraction, then t is a fireball.

Let t be an application, *i.e.* $t = us$. By *i.h.*, there are two cases for s:

(a) *s has exactly one $\mathtt{r}\beta_f$-redex*. Then t has an $\mathtt{r}\beta_f$-redex, because $u\langle\cdot\rangle$ is a right context. Moreover, no $\mathtt{r}\beta_f$-redex in t can lie in u, and t itself is not an $\mathtt{r}\beta_f$-redex, since by open harmony (Proposition 1.1) s is not a fireball and so $\langle\cdot\rangle s$ is not a right context. Thus, t has exactly one $\mathtt{r}\beta_f$-redex.
(b) *s has no $\mathtt{r}\beta_f$-redexes*. By *i.h.*, there are two cases for u:

i. *u has exactly one $\mathtt{r}\beta_f$-redex*. Then t has an $\mathtt{r}\beta_f$-redex, because $\langle\cdot\rangle s$ is a right context as s is a fireball. Uniqueness follows from the fact that s has no $\mathtt{r}\beta_f$-redexes and u is not an abstraction.
ii. *u has no $\mathtt{r}\beta_f$-redexes*. So, u is a fireball. There are two sub-cases:

- *u is an abstraction $\lambda x.r$*. Then $t = (\lambda x.r)s$ is an $\mathtt{r}\beta_f$-redex, because s is a fireball. Also, there are no other $\mathtt{r}\beta_f$-redexes, as right contexts do not enter abstractions and s is a fireball.
- *u is an inert term*. Since s is a fireball, t is so (as inert) and hence t has no $\mathtt{r}\beta_f$-redexes. □

Example 1. Let $t := (\lambda z.z(yz))\lambda x.x$ with $z \neq y$. Then, $t \to_{\mathtt{r}\beta_f} (\lambda x.x)(y\,\lambda x.x) \to_{\mathtt{r}\beta_f} y\,\lambda x.x$, where the final term $y\,\lambda x.x$ is a fireball (and β_f-normal), since it is an inert term.

Right-to-left vs. left-to-right. In implementing Open CbV, an asymmetry between right-to-left and left-to-right evaluations arises.

In Closed CbV, the two strategies require slightly different but essentially identical abstract machines. The reason is that in the closed case β_λ-redexes are symmetric: both the left and right subterms are abstractions, and abstractions are recognized in constant time by looking only at the topmost constructor.

Switching to the open case, the left-to-right strategy becomes a bit trickier to implement. Indeed, β_f-redexes are asymmetric, since the right subterm may be an inert term—inert terms have a more complex structure than abstractions, and they are not recognizable in constant time. Consequently, a left-to-right machine checks the easy part first and the complex second, needing a *backtracking phase* after the complex check to come back to the original redex. This mechanism is akin to that of machines for strong evaluation such as the Strong MAM in [13]. The right-to-left machine instead does the complex part first, and so it does not need to backtrack—this simplicity is why we work with the right-to-left strategy.

Open size explosion. Fireballs are delicate, they can easily *explode*. The simplest instance of *open size explosion* (not existing in Closed CbV) is a variation over the famous looping term $\omega := (\lambda x.xx)(\lambda x.xx) \to_{\beta_\lambda} \omega \to_{\beta_\lambda} \dots$. In ω there is an infinite sequence of duplications. In the size exploding family there is a sequence of n nested duplications. We define two families, the family $\{t_n\}_{n\in\mathbb{N}}$ of size exploding terms and the family $\{i_n\}_{n\in\mathbb{N}}$ of results of evaluating $\{t_n\}_{n\in\mathbb{N}}$:

$$t_0 := y \qquad t_{n+1} := (\lambda x.xx)t_n \qquad\qquad i_0 := y \qquad i_{n+1} := i_n i_n.$$

The *size* $|t|$ of a term t is defined inductively as expected (roughly, it counts the number of symbols in t; more precisely, it counts the number of nodes in the tree representation of t): $|x| := 1$ and $|\lambda x.t| := |t| + 1$ and $|tu| := |t| + |u| + 1$.

Proposition 3 *(Open size explosion, [10]). Let $n \in \mathbb{N}$. Then $t_n \to_{\beta_i}^{n} i_n$, moreover $|t_n| = O(n)$, $|i_n| = \Omega(2^n)$, and i_n is an inert term (and so β_f-normal).*

Proof. By induction on n. The base case is trivial, as $t_0 = y = i_0$. Inductive case: $t_{n+1} = (\lambda x.xx)t_n \to_{\beta_i}^{n} (\lambda x.xx)i_n \to_{\beta_i} i_n i_n = i_{n+1}$, where the first sequence is obtained by the *i.h.* Clearly i_{n+1} is inert, and the bounds on the sizes are immediate. □

Circumventing open size explosion. Abstract machines implementing the substitution of inert terms, such as the one described by Grégoire and Leroy [27] (discussed here in Sect. 4, where it is called Open GLAM) are unreasonable because for the term t_n of the size exploding family in Proposition 3 they compute the full result i_n, whose size is exponential in n. The various GLAMOUr machines of the next sections are instead reasonable because they *avoid the substitution of inert terms*, which is justified by the fact that *evaluating* and *substituting inert terms* are operations that commute, as Proposition 4 below shows. In order to prove that, we need the next two technical lemmas.

Lemma 2 *(Fireballs are closed under substitution and anti-substitution of inert terms).*[3] *Let t be a term and i be an inert term.*

1. *$t\{x\leftarrow i\}$ is an abstraction if and only if t is an abstraction;*
2. *$t\{x\leftarrow i\}$ is an inert term if and only if t is an inert term;*
3. *$t\{x\leftarrow i\}$ is a fireball if and only if t is a fireball.*

Proof. Point 1 is trivial. Point 3 is an immediate consequence of Points 1-2.

Concerning Point 2, the left-to-right direction ($\Rightarrow$) is proved by a simple induction on the inert structure of $t\{x\leftarrow i\}$. Conversely, the right-to-left direction ($\Leftarrow$) is proved by a simple induction on the inert structure of t. □

Lemma 3 *(Substitution of inert terms does not create β_f-redexes).*[4] *Let t, u be terms and i be an inert term. There is a term s such that:*

1. *if $t\{x\leftarrow i\} \to_{\beta_\lambda} u$ then $t \to_{\beta_\lambda} s$ and $s\{x\leftarrow i\} = u$;*
2. *if $t\{x\leftarrow i\} \to_{\beta_i} u$ then $t \to_{\beta_i} s$ and $s\{x\leftarrow i\} = u$.*

Proof. By induction on the definition of $t\{x\leftarrow i\} \to_{\beta_f} u$ (*i.e.* on the evaluation context closing the root redex). □

We can now prove that *evaluation and substitution of inert terms commute*. Said differently, substitution of inert terms can always be postponed and hence safely avoided.

Proposition 4 *(Inert substitutions and evaluation commute). Let t, u be terms and i be an inert term. Then, $t \to_{\beta_f} u$ if and only if $t\{x\leftarrow i\} \to_{\beta_f} u\{x\leftarrow i\}$. More precisely:*

1. Inert substitutions do not erase β_f-redexes: *If $t \to_{\beta_f} u$ then $t\{x\leftarrow i\} \to_{\beta_f} u\{x\leftarrow i\}$.*
2. Inert substitutions do not create β_f-redexes: *If $t\{x\leftarrow i\} \to_{\beta_f} u$ then there is a term t' such that $t \to_{\beta_f} t'$ and $u = t'\{x\leftarrow i\}$.*

Proof. Point 2 is exactly Lemma 3, since $\to_{\beta_f} = \to_{\beta_\lambda} \cup \to_{\beta_i}$.

Point 1 is proved by induction on the definition of $t \to_{\beta_f} u$. Cases:

- *Step at the root*:
  1. *Abstraction step*, *i.e.* $t = (\lambda y.s)r \mapsto_{\beta_\lambda} s\{y\leftarrow r\} = u$ where r is an abstraction (assume without loss of generality $y \notin \mathtt{fv}(i) \cup \{x\}$). Since $r\{x\leftarrow i\}$ is an abstraction (Lemma 2.1), $t\{x\leftarrow i\} = (\lambda y.s\{x\leftarrow i\})r\{x\leftarrow i\} \mapsto_{\beta_\lambda} s\{x\leftarrow i\}\{y\leftarrow r\{x\leftarrow i\}\} = s\{y\leftarrow r\}\{x\leftarrow i\} = u\{x\leftarrow i\}$.
  2. *Inert step*, *i.e.* $t = (\lambda y.s)i' \mapsto_{\beta_i} s\{y\leftarrow i'\} = u$ where i' is inert. We can suppose without loss of generality that $y \notin \mathtt{fv}(i) \cup \{x\}$. According to Lemma 2.2, $i'\{x\leftarrow i\}$ is inert. So, $t\{x\leftarrow i\} = (\lambda y.s\{x\leftarrow i\})i'\{x\leftarrow i\} \mapsto_{\beta_i} s\{x\leftarrow i\}\{y\leftarrow i'\{x\leftarrow i\}\} = s\{y\leftarrow i'\}\{x\leftarrow i\} = u\{x\leftarrow i\}$.
- *Application right*, *i.e.* $t = rs \to_{\beta_f} rs' = u$ with $s \to_{\beta_f} s'$. By *i.h.*, $s\{x\leftarrow i\} \to_{\beta_f} s'\{x\leftarrow i\}$, so $t\{x\leftarrow i\} = r\{x\leftarrow i\}s\{x\leftarrow i\} \to_{\beta_f} r\{x\leftarrow i\}s'\{x\leftarrow i\} = u\{x\leftarrow i\}$.
- *Application left*, *i.e.* $t = sr \to_{\beta_f} s'r = u$ with $s \to_{\beta_f} s'$. Analogous to the *application right* case, just switch right and left. □

Proposition 4 states that substitution of inert terms for variables cannot create or erase β_f-redexes, which is why it can be avoided. This property is distinctive of inert terms (hence their name). With general terms (or even fireballs) instead of inert ones, evaluation and substitution do not commute, that is both points of Proposition 4 do not hold. Point 2 is false

[3] Proof in Appendix, p. 27.
[4] Proof in Appendix, p. 28.

because substitution can create β_f-redexes, as in $(xy)\{x\leftarrow\lambda z.z\} = (\lambda z.z)y$; Point 1 is false because substitution can erase β_f-redexes, as in $((\lambda x.z)(xx))\{x\leftarrow\delta\} = (\lambda x.z)(\delta\delta)$ where $\delta \coloneqq \lambda y.yy$.[5]

Variables and inert terms. Variables can be seen both as values and inert terms. Here we consider them as inert terms, because for abstract machines it is practical to see values as being only abstractions. The substitution of general inert terms causes open size explosion, but of course the substitution of variables is harmless. And whether variables are substituted or not is one of the parameters that will play a role in our study of abstract machines, in the following sections.

3. Abstract machines, implementations, and complexity analyses

In this section we introduce general notions about abstract machines, given with respect to a generic machine $\mathtt{M}$ and a generic strategy $\to$ on λ-terms. Then we give an abstract notion of implementation and sufficient conditions for it. Finally, we provide a general recipe for complexity analyses.

Abstract machines glossary.

- An abstract machine $\mathtt{M}$ is given by *states*, noted s, and *transitions* between them, noted $\leadsto_{\mathtt{M}}$; the reflexive-transitive closure of $\leadsto_{\mathtt{M}}$ is noted $\leadsto_{\mathtt{M}}^*$;
- A state is given by the *code under evaluation* plus some *data structures*, which can be seen as lists of items (the cons operator for such lists is denoted by $:$);
- The code under evaluation, as well as the other pieces of code scattered in the data structures, are λ-terms *not considered modulo α-equivalence*; they are overlined, to stress the different treatment of α-equivalence; the *size* $|\overline{t}|$ of a code $\overline{t}$ is exactly the size $|t|$ of any term t α-equivalent to $\overline{t}$;
- A code $\overline{t}$ is *well-named* if, for every sub-code $\lambda x.\overline{u}$ of $\overline{t}$, the variable x may occur only in $\overline{u}$ (if at all);
- A state is *initial* if its code is well-named and its data structures are empty (an *empty list* of items is denoted by ϵ);
- Therefore, there is a bijection $\cdot^\circ$ (up to α-equivalence) between λ-terms and initial states, called *compilation*, sending a λ-term t to the initial state t° on a well-named code α-equivalent to t;
- An *execution* is a finite (possibly empty) sequence of transitions $t_0^\circ \leadsto_{\mathtt{M}}^* s$ from an initial state t_0° obtained by compiling an (initial) λ-term t_0;
- A state s is *reachable* if there is an execution $t_0^\circ \leadsto_{\mathtt{M}}^* s$, for some λ-term t_0;
- A state s is *final* if it is reachable and no transitions apply to s;
- A machine comes with a *decoding* map $\underline{\cdot}$ from states to λ-terms that on initial states is the inverse (up to α-equivalence) of compilation, *i.e.* $\underline{t^\circ} = t$ for any λ-term t, and so $(\underline{s})^\circ = s$ for any initial state s (as it is of the form $s = t^\circ$);
- Transitions of a machine $\mathtt{M}$ are divided into *β-transitions*, noted $\leadsto_\beta$, which are meant to be mapped to β-reduction steps by the decoding, while the remaining *overhead transitions*, noted $\leadsto_\circ$, are mapped to equalities;
- We use $|\rho|$ for the length (*i.e.* number of transitions) of an execution ρ, and $|\rho|_\beta$ for the number of β-transitions in ρ.

Implementations. Any machine has to be proven to implement correctly the strategy in the λ-calculus for which it is conceived. Our notion of implementation, tuned towards complexity analyses, claims a *perfect match* between the number of β-steps of the strategy and the number of β-transitions of the machine execution.

Definition 1 *(Machine implementation).* An abstract machine $\mathtt{M}$ *implements a strategy* $\to$ on λ-terms via a decoding $\underline{\cdot}$ when, for every λ-term t, the following hold:

1. *Executions to derivations*: for any $\mathtt{M}$-execution $\rho : t^\circ \leadsto_{\mathtt{M}}^* s$ there exists a $\to$-derivation $d : t \to^* \underline{s}$.
2. *Derivations to executions*: for every $\to$-derivation $d : t \to^* u$ there exists a $\mathtt{M}$-execution $\rho : t^\circ \leadsto_{\mathtt{M}}^* s$ such that $\underline{s} = u$.
3. *β-matching*: in both previous points the number $|\rho|_\beta$ of β-transitions in ρ is exactly the length $|d|$ of d, *i.e.* $|d| = |\rho|_\beta$.

Sufficient condition for implementations. The proofs that some machine implements a strategy tend to follow always the same structure, based on a few abstract properties collected here into the notion of implementation system.

Definition 2 *(Implementation system).* An abstract machine $\mathtt{M}$, a strategy $\to$, and a decoding $\underline{\cdot}$ form an *implementation system* if the following conditions hold:

1. *β-projection*: $s \leadsto_\beta s'$ implies $\underline{s} \to \underline{s'}$, for every reachable state s;
2. *Overhead transparency*: $s \leadsto_\circ s'$ implies $\underline{s} = \underline{s'}$, for every reachable state s;

[5] As well-known in the theory of λ-calculus, Proposition 4 with ordinary (*i.e.* call-by-name) β-reduction $\to_\beta$ instead of $\to_{\beta_f}$ and general terms instead of inert ones holds only in Point 1.

3. *Overhead termination*: $\rightsquigarrow_{\circ}$ terminates, starting from every reachable state;
4. *Determinism*: both $\rightsquigarrow_{\mathtt{M}}$ and $\to$ are deterministic;
5. *Halt*: $\mathtt{M}$ final states decode to $\to$-normal terms.

Now we prove the implementation theorem (Theorem 1), stating that the conditions required to an implementation system (Definition 2) indeed imply that the machine implements the strategy via the decoding (in the sense of Definition 1).

The *executions-to-derivations* part of the implementation theorem is easy to prove, essentially *β-projection* and *overhead transparency* allow us to project a single transition onto one or none $\to$-steps, and the projection of executions onto derivations is obtained as a simple induction.

The *derivations-to-executions* part is a bit more delicate, instead, because the simulation of $\to$-steps into the machine has to be done *up to* overhead transitions. The following lemma shows how the conditions for implementation systems allow us to do that. Interestingly, all five conditions of Definition 2 are used in the proof.

Lemma 4 *(One-step simulation). Let $\mathtt{M}$, $\to$, and $\underline{\cdot}$ be a machine, a strategy, and a decoding forming an implementation system. For any reachable state s of $\mathtt{M}$, if $\underline{s} \to u$ then there is a state s' of $\mathtt{M}$ such that $s \rightsquigarrow_{\circ}^{*} \rightsquigarrow_{\beta} s'$ and $\underline{s'} = u$.*

Proof. For any reachable state s of $\mathtt{M}$, let $\mathsf{nf}_{\circ}(s)$ be the normal form of s with respect to $\rightsquigarrow_{\circ}$: such a state exists and is unique because overhead transitions terminate (Point 3) and $\mathtt{M}$ is deterministic (Point 4). Since $\rightsquigarrow_{\circ}$ is mapped on identities (Point 2), one has $\underline{\mathsf{nf}_{\circ}(s)} = \underline{s}$. As $\underline{s}$ is not $\to$-normal by hypothesis, the halt property (Point 5) entails that $\mathsf{nf}_{\circ}(s)$ is not final, therefore $s \rightsquigarrow_{\circ}^{*} \mathsf{nf}_{\circ}(s) \rightsquigarrow_{\beta} s'$ for some state s', and thus $\underline{s} = \underline{\mathsf{nf}_{\circ}(s)} \to \underline{s'}$ by β-projection (Point 1). According to the determinism of $\to$ (Point 4), one obtains $\underline{s'} = u$. □

The one-step simulation (Lemma 4) can be extended to the simulation of derivations into the machine by an easy induction on the length of the derivation.

Theorem 1 *(Sufficient condition for implementations). Let $(\mathtt{M}, \to, \underline{\cdot})$ be an* implementation system. *Then, $\mathtt{M}$ implements $\to$ via $\underline{\cdot}$.*

Proof. According to Definition 1, given a λ-term t, we have to show that:

(i) *Executions to derivations with β-matching*: for any $\mathtt{M}$-execution $\rho : t^{\circ} \rightsquigarrow_{\mathtt{M}}^{*} s$ there exists a $\to$-derivation $d : t \to^{*} \underline{s}$ such that $|d| = |\rho|_{\beta}$.
(ii) *Derivations to executions with β-matching*: for any $\to$-derivation $d : t \to^{*} u$ there exists a $\mathtt{M}$-execution $\rho : t^{\circ} \rightsquigarrow_{\mathtt{M}}^{*} s$ such that $\underline{s} = u$ and $|d| = |\rho|_{\beta}$.

Proof of Point (i). By induction on $|\rho|_{\beta} \in \mathbb{N}$.

If $|\rho|_{\beta} = 0$ then $\rho : t^{\circ} \rightsquigarrow_{\circ}^{*} s$ and hence $\underline{t^{\circ}} = \underline{s}$ by overhead transparency (Point 2 of Definition 2). Moreover, $t = \underline{t^{\circ}}$ since decoding is the inverse of compilation on initial states, therefore we are done by taking the empty (*i.e.* without steps) derivation d with starting (and end) term t.

Suppose $|\rho|_{\beta} > 0$: then, $\rho : t^{\circ} \rightsquigarrow_{\mathtt{M}}^{*} s$ is the concatenation of an execution $\rho' : t^{\circ} \rightsquigarrow_{\mathtt{M}}^{*} s'$ followed by an execution $\rho'' : s' \rightsquigarrow_{\beta} s'' \rightsquigarrow_{\circ}^{*} s$. By *i.h.* applied to ρ', there exists a derivation $d' : t \to^{*} \underline{s'}$ with $|\rho'|_{\beta} = |d'|$. By β-projection (Point 1 of Definition 2) and overhead transparency (Point 2 of Definition 2) applied to ρ'', one has $d'' : \underline{s'} \to \underline{s''} = \underline{s}$. Therefore, the derivation d defined as the concatenation of d' and d'' is such that $d : t \to^{*} \underline{s}$ and $|d| = |d'| + |d''| = |\rho'|_{\beta} + 1 = |\rho|_{\beta}$.

Proof of Point (ii). By induction on $|d| \in \mathbb{N}$.

If $|d| = 0$ then $t = u$. Since decoding is the inverse of compilation on initial states, one has $\underline{t^{\circ}} = t$. We are done by taking the empty (*i.e.* without transitions) execution ρ with initial (and end) state t°.

Suppose $|d| > 0$: so, $d : t \to^{*} u$ is the concatenation of a derivation $d' : t \to^{*} u'$ followed by the step $u' \to u$. By *i.h.*, there exists a $\mathtt{M}$-execution $\rho' : t^{\circ} \rightsquigarrow_{\mathtt{M}}^{*} s'$ such that $\underline{s'} = u'$ and $|d'| = |\rho'|_{\beta}$. According to the one-step simulation (Lemma 4, since $\underline{s'} \to u$ and $(\mathtt{M}, \to, \underline{\cdot})$ is an implementation system), there is a state s of $\mathtt{M}$ such that $s' \rightsquigarrow_{\circ}^{*} \rightsquigarrow_{\beta} s$ and $\underline{s} = u$. Therefore, the execution $\rho : t^{\circ} \rightsquigarrow_{\mathtt{M}}^{*} s' \rightsquigarrow_{\circ}^{*} \rightsquigarrow_{\beta} s$ is such that $|\rho|_{\beta} = |\rho'|_{\beta} + 1 = |d'| + 1 = |d|$. □

The proof of Theorem 1 is a clean and abstract generalization of the concrete reasoning used in [12,10,2,3,14] for specific abstract machines and strategies, and it is a contribution of this work.

Parameters for complexity analyses. Let $\mathtt{M}$ be an abstract machine implementing a strategy $\to$ via a decoding $\underline{\cdot}$ (Definition 1). By the *derivations-to-executions* part of the implementation (Point 2 in Definition 1), given a derivation $d : t_0 \to^{n} u$ there is a shortest execution $\rho : t_0^{\circ} \rightsquigarrow_{\mathtt{M}}^{*} s$ such that $\underline{s} = u$. Determining *the complexity of a machine* $\mathtt{M}$ amounts to bound the asymptotic complexity of a concrete implementation of ρ on a random access machine (RAM), as a function of *two* parameters:

1. *Input*: the size $|t_0|$ of the initial term t_0 of the derivation d;

2. β-*steps/transitions*: the length $n = |d|$ of the derivation d, that coincides with the number $|\rho|_\beta$ of β-transitions in ρ by the β-matching requirement for implementations (Point 3 in Definition 1).

A machine is *reasonable* if its complexity is polynomial in $|t_0|$ and $|\rho|_\beta$, and it is *efficient* if it is linear in both parameters. So, a strategy is reasonable (resp. efficient) if there is a reasonable (resp. efficient) machine implementing it. In Sect. 5-6 we study a reasonable machine implementing right-to-left evaluation $\to_{\mathtt{r}\beta_f}$ in λ_{fire}, thus showing that it is a reasonable strategy. In Sect. 7 we optimize the machine to make it efficient. By Proposition 2, this implies that *every* strategy in λ_{fire} is efficient.

Recipe for complexity analyses. In a machine $\mathtt{M}$, overhead transitions $\rightsquigarrow_{\mathtt{o}}$ are further separated into two classes:

1. *Substitution transitions* $\rightsquigarrow_{\mathtt{s}}$: they are in charge of the substitution process;
2. *Commutative transitions* $\rightsquigarrow_{\mathtt{c}}$: they are in charge of searching for the next β or substitution redex to reduce.

Then, the estimation of the complexity of a machine is done in three steps:

1. *Number of transitions*: bounding the length $|\rho|$ of the execution ρ, by bounding the number of its overhead transitions. Both the number $|\rho|_{\mathtt{c}}$ of commutative transitions and the number $|\rho|_{\mathtt{s}}$ of substitution transitions are—separately—bounded using the size $|t_0|$ of the input t_0 and the number $|\rho|_\beta$ of β-transitions in ρ. For some machines, the bound on $|\rho|_{\mathtt{s}}$ turns out to depend only on $|\rho|_\beta$.
2. *Cost of single transitions*: bounding the cost of concretely implementing a single transition of $\mathtt{M}$. Here it is necessary to go beyond the abstract level, making some (high-level) assumption about how code and data structures are concretely represented. Commutative transitions are designed on purpose to have constant cost. Each substitution transition has a cost linear in the size of the initial term thanks to an invariant (to be proved) ensuring that only subterms of the initial term are duplicated and substituted along an execution. Each β-transition has a cost constant or linear in the input.
3. *Complexity of the overhead*: obtaining the total bound by composing the first two points, that is, by taking the number of each kind of transition times the cost of implementing it, and summing over all kinds of transitions.

(Linear) logical reading. Our partitioning of transitions into β, substitution and commutative ones admits a proof-theoretical view, as machine transitions can be seen as cut-elimination steps [15,12]. Substitution and β transitions correspond to principal cases in cut-elimination. Moreover, in linear logic the β-transition corresponds to the multiplicative case while the substitution transition to the exponential one. See [12] for more details.

4. Open GLAM

In this section we quickly recall the *Open GLAM* from Accattoli and Sacerdoti Coen [10], *i.e.* the core (up to syntactic sugar) of the abstract machine introduced by Grégoire and Leroy [27] to improve the implementation of Coq. The Open GLAM is the natural—and yet naïve—way to implement the right-to-left strategy $\to_{\mathtt{r}\beta_f}$ of the fireball calculus λ_{fire}.

As we will see, the Open GLAM is an *unreasonable* implementation of the fireball calculus, because its overhead is exponential in the number of β_f-steps—essentially, it does not circumvent open size explosion (Proposition 3). For this reason, we omit a formal study of the properties of the Open GLAM, preserving our technical energies for the reasonable machines of the next sections. We present it anyway, to help the reader become familiar with abstract machines, and because the other machines have the same data structures as the Open GLAM and transitions that are obtained by simple tweaks of the Open GLAM.

The name Open GLAM comes from the Leroy Abstract Machine (LAM), a similar machine implementing Closed CbV introduced in [12]. The adjective *Open* is due to the fact that the machine handles also open terms. The *G* in GLAM instead is due to the use of *global* (rather than local) environments.

Data structures. The machines in this paper are unusual in two respects.

First and more importantly, they use a single *global* environment instead of *local environments* and *closures*. Global environments are used in a few papers [23,33,22,12,10,13,2] and induce simpler, more abstract machines where α-equivalence is pushed to the meta-level (the operation $\overline{t}^{\alpha}$ in the substitution transition $\rightsquigarrow_{\mathtt{s}}$ for the machines in Tables 3–5, 8–9). This on-the-fly α-renaming is harmless with respect to complexity analyses. For a thorough comparison of different approaches to environments and of how they impact on the implementation complexity, see Accattoli and Barras [5].

Second, argument stacks contain *pairs* of a code and a stack, to implement some of the machine transitions in constant time, namely the commutative ones.

The configuration in a given time of the Open GLAM (and of the machines in the GLAMOUr family) is stored in a *state*, formally defined in Table 2 as a quadruple $s = (D, \overline{t}, \pi, E)$ of the following data structures:

- *Code* $\overline{t}$: a term *not* considered up to α-equivalence, this is why t is overlined.
- *Argument stack* π: a list containing the arguments of the current code. Note that stack items ϕ are pairs of the form $x@\pi$ or $\lambda x.\overline{u}@\epsilon$. These pairs allow us to implement some of the transitions in constant time. The pair $x@\pi'$ (where π' is

Table 2
(On the left) *Data-structures* for the Open GLAM (Table 3) and the machines of the GLAMOUr family (Tables 4–9): *items* ϕ, *stacks* π, *dumps* D, *global environments* E, *states* s. (On the right) *Unfolding* $t\downarrow_E$ and *decoding* $\underline{\cdot}$ (stacks are decoded to contexts in postfix notation for plugging, *i.e.* we write $\langle\bar{t}\rangle\underline{\pi}$ rather than $\underline{\pi}\langle\bar{t}\rangle$).

Data structures		Decoding & Unfolding	
Stack item	$\phi,\psi ::= \lambda x.\overline{u}@\epsilon \mid x@\pi$	$\underline{\epsilon} := \langle\cdot\rangle$	$t\downarrow_\epsilon := t \quad t\downarrow_{[x\leftarrow\phi]E} := t\{x\leftarrow\underline{\phi}\}\downarrow_E$
Stack	$\pi ::= \epsilon \mid \phi : \pi$	$\underline{\phi:\pi} := \langle\langle\cdot\rangle\underline{\phi}\rangle\underline{\pi}$	$\langle\cdot\rangle\downarrow_E := \langle\cdot\rangle \quad (Ct)\downarrow_E := C\downarrow_E t\downarrow_E \quad (tC)\downarrow_E := t\downarrow_E C\downarrow_E$
Environment	$E ::= \epsilon \mid [x\leftarrow\phi]:E$	$\underline{\bar{t}@\pi} := \langle\bar{t}\rangle\underline{\pi}$	$R_s := \underline{D}\langle\underline{\pi}\rangle\downarrow_E$ where $s = (D,\bar{t},\pi,E)$
Dump	$D ::= \epsilon \mid D : \bar{t}\diamondsuit\pi$	$\underline{D:\bar{t}\diamondsuit\pi} := \underline{D}\langle\langle\bar{t}\langle\cdot\rangle\rangle\underline{\pi}\rangle$	$\underline{s} := \underline{D}\langle\langle\bar{t}\rangle\underline{\pi}\rangle\downarrow_E = R_s\langle\bar{t}\downarrow_E\rangle$ where $s = (D,\bar{t},\pi,E)$
State	$s := (D,\bar{t},\pi,E)$		

Table 3
Transitions of the Open GLAM. In the substitution transition $\rightsquigarrow_s$, $(\underline{\phi})^\alpha$ is any well-named code α-equivalent to $\underline{\phi}$ such that its bound variables are fresh with respect to those in D, π and $E[x\leftarrow\phi]E'$.

Dump	Code	Stack	Global Env.		Dump	Code	Stack	Global Env.
D	$\bar{t}\bar{u}$	π	E	$\rightsquigarrow_{c_1}$	$D:\bar{t}\diamondsuit\pi$	$\bar{u}$	ϵ	E
$D:\bar{t}\diamondsuit\pi$	$\lambda x.\bar{u}$	ϵ	E	$\rightsquigarrow_{c_2}$	D	$\bar{t}$	$(\lambda x.\bar{u}@\epsilon):\pi$	E
$D:\bar{t}\diamondsuit\pi$	x	π'	E	$\rightsquigarrow_{c_3}$	D	$\bar{t}$	$(x@\pi'):\pi$	E
								if $E(x)=\bot$
D	$\lambda x.\bar{t}$	$\phi:\pi$	E	$\rightsquigarrow_\beta$	D	$\bar{t}$	π	$[x\leftarrow\phi]E$
D	x	π	$E[x\leftarrow\phi]E'$	$\rightsquigarrow_s$	D	$(\underline{\phi})^\alpha$	π	$E[x\leftarrow\phi]E'$

in turn an argument stack) codes the inert term $\langle x\rangle\underline{\pi'}$ (defined in Table 2—the decoding $\underline{\cdot}$ is explained below) obtained by putting x in the context obtained by decoding π'. The pair $\lambda x.\overline{u}@\epsilon$ (where ϵ is the empty stack) is used to inject abstractions into pairs, so that items ϕ can be uniformly seen as pairs $\bar{t}@\pi$ of a code $\bar{t}$ and a stack π.

- *Dump* D: a second stack that, together with the argument stack π, is used to walk through the code and search for the next redex to reduce. The dump is extended (on the right) with an entry $\bar{t}\diamondsuit\pi$ every time evaluation enters the right subterm $\overline{u}$ of an application $\bar{t}\overline{u}$; the entry saves the left part $\bar{t}$ of the application and the current stack π, to restore them when the evaluation of $\overline{u}$ is over. The dump D and the stack π decode to a right context.
- *Global environment* E: a list of explicit (*i.e.* delayed) substitutions storing the β-redexes encountered so far. It is used to implement micro-step evaluation (substitution for one variable occurrence at a time). We write $E(x)=\bot$ if E has no entries of the form $[x\leftarrow\phi]$. Often $[x\leftarrow\phi]E$ stands for $[x\leftarrow\phi]:E$, and by abuse of notation we confuse cons and concatenation. Note that the body ϕ of any environment entry $[x\leftarrow\phi]$ is a stack item, not a code.

Transitions. The Open GLAM has one β-transition whereas overhead transitions are divided up into substitution and commutative transitions, see Table 3.

- *β-transition* $\rightsquigarrow_\beta$: it morally fires the $\mathtt{r}\beta_f$-redex corresponding to $(\lambda x.\bar{t})\phi$, except that it puts a new delayed substitution $[x\leftarrow\phi]$ in the environment instead of doing the meta-level substitution $\bar{t}\{x\leftarrow\phi\}$ of the argument ϕ for the (free) occurrences of the variable x in the body $\bar{t}$ of the abstraction.
- *Substitution transition* $\rightsquigarrow_s$: it substitutes the variable occurrence under evaluation with an (α-renamed copy of a) code from the environment. It is a micro-step (*i.e.* linear, in the sense of one occurrence at a time) variant of meta-level substitution. It is invisible on λ_{fire}: the decoding produces the term obtained by meta-level substitution, so the micro work done by $\rightsquigarrow_s$ cannot be observed at the coarser granularity of λ_{fire}. Note that $\rightsquigarrow_s$ is triggered whenever the current code is a variable bound in the environment to *any* item ϕ: as ϕ is a pair code–stack, $\rightsquigarrow_s$ must decode ϕ too (see below).
- *Commutative transitions* $\rightsquigarrow_c$: they locate and expose the next $\mathtt{r}\beta_f$-redex according to the right-to-left strategy, by rearranging the data structures. They are invisible on the calculus. The transition $\rightsquigarrow_{c_1}$ forces evaluation to be right-to-left on applications $\bar{t}\overline{u}$: the machine processes first the right subterm $\overline{u}$, saving the left sub-term $\bar{t}$ on the dump along with its current stack π. The role of $\rightsquigarrow_{c_2}$ and $\rightsquigarrow_{c_3}$ is to backtrack to the entry on top of the dump. When the right subterm, *i.e.* the pair $\overline{u}@\pi'$ of current code and stack, is finally in normal form, it is pushed on the stack and the machine backtracks. Condition $E(x)=\bot$ (which means that the variable x is not bound) in $\rightsquigarrow_{c_3}$ is how the Open GLAM handles open terms.

Note the absence of *garbage collection*: it is here simply ignored, or, more precisely, it is encapsulated at the meta-level, in the decoding function $\underline{\cdot}$. It is well-known that this is harmless for the study of time complexity.

Compiling and decoding. A term t is compiled to the machine *initial state* $t^\circ = (\epsilon,\bar{t},\epsilon,\epsilon)$, where $\bar{t}$ is a well-named code α-equivalent to t. Conversely, every machine state $s=(D,\bar{t},\pi,E)$ decodes to a term $\underline{s}$ (see the right part of Table 2), having the shape $R_s\langle\bar{t}\downarrow_E\rangle$, where $\bar{t}\downarrow_E$ is the λ-term (called *unfolding*) obtained by recursively substituting—at the meta-level—the entries of the global environment E on $\bar{t}$, and R_s is a right context, obtained by decoding the stack π and the dump D and then applying the unfolding $\downarrow_E$. To improve readability, stacks are decoded to contexts in postfix notation for plugging, *i.e.* we write $\langle\bar{t}\rangle\underline{\pi}$ rather than $\underline{\pi}\langle\bar{t}\rangle$ because π is a context that puts arguments in front of $\bar{t}$. Note that $\underline{t^\circ}=t$ for any term t.

Unreasonable implementation. The Open GLAM implements (in the sense of Definition 2) the right-to-left evaluation $\to_{\mathtt{r}\beta_f}$ of the fireball calculus λ_{fire}, via the decoding $\underline{\cdot}$, as shown in [10]. That work also shows that the Open GLAM is unreasonable since it is affected by the open size explosion problem (Proposition 3).

Example 2. To have a glimpse of how the Open GLAM works, we show its execution on one of the terms (for $n = 2$) of the open size exploding family of Proposition 3, namely $t_2 := (\lambda x_1.x_1x_1)t_1 = (\lambda x_1.x_1x_1)((\lambda x_0.x_0x_0)y) \to^2_{\beta_i} (yy)(yy) =: i_2$.

Dump	Code	Stack	Global Environment	
ϵ	$(\lambda x_1.x_1x_1)t_1$	ϵ	ϵ	$\rightsquigarrow_{c_1}$
$\lambda x_1.x_1x_1 \Diamond \epsilon$	$(\lambda x_0.x_0x_0)y$	ϵ	ϵ	$\rightsquigarrow_{c_1}$
$\lambda x_1.x_1x_1 \Diamond \epsilon : \lambda x_0.x_0x_0 \Diamond \epsilon$	y	ϵ	ϵ	$\rightsquigarrow_{c_3}$
$\lambda x_1.x_1x_1 \Diamond \epsilon$	$\lambda x_0.x_0x_0$	$y@\epsilon$	ϵ	$\rightsquigarrow_{\beta}$
$\lambda x_1.x_1x_1 \Diamond \epsilon$	x_0x_0	ϵ	$[x_0 \leftarrow y@\epsilon]$	$\rightsquigarrow_{c_1}$
$\lambda x_1.x_1x_1 \Diamond \epsilon : x_0 \Diamond \epsilon$	x_0	ϵ	$[x_0 \leftarrow y@\epsilon]$	$\rightsquigarrow_{s}$
$\lambda x_1.x_1x_1 \Diamond \epsilon : x_0 \Diamond \epsilon$	y	ϵ	$[x_0 \leftarrow y@\epsilon]$	$\rightsquigarrow_{c_3}$
$\lambda x_1.x_1x_1 \Diamond \epsilon$	x_0	$y@\epsilon$	$[x_0 \leftarrow y@\epsilon]$	$\rightsquigarrow_{s}$
$\lambda x_1.x_1x_1 \Diamond \epsilon$	y	$y@\epsilon$	$[x_0 \leftarrow y@\epsilon]$	$\rightsquigarrow_{c_3}$
ϵ	$\lambda x_1.x_1x_1$	$y@(y@\epsilon)$	$[x_0 \leftarrow y@\epsilon]$	$\rightsquigarrow_{\beta}$
ϵ	x_1x_1	ϵ	$[x_1 \leftarrow y@(y@\epsilon)] : [x_0 \leftarrow y@\epsilon]$	$\rightsquigarrow_{c_1}$
$x_1 \Diamond \epsilon$	x_1	ϵ	$[x_1 \leftarrow y@(y@\epsilon)] : [x_0 \leftarrow y@\epsilon]$	$\rightsquigarrow_{s}$
$x_1 \Diamond \epsilon$	yy	ϵ	$[x_1 \leftarrow y@(y@\epsilon)] : [x_0 \leftarrow y@\epsilon]$	$\rightsquigarrow_{c_1}$
$x_1 \Diamond \epsilon : y \Diamond \epsilon$	y	ϵ	$[x_1 \leftarrow y@(y@\epsilon)] : [x_0 \leftarrow y@\epsilon]$	$\rightsquigarrow_{c_3}$
$x_1 \Diamond \epsilon$	y	$y@\epsilon$	$[x_1 \leftarrow y@(y@\epsilon)] : [x_0 \leftarrow y@\epsilon]$	$\rightsquigarrow_{c_3}$
ϵ	x_1	$y@(y@\epsilon)$	$[x_1 \leftarrow y@(y@\epsilon)] : [x_0 \leftarrow y@\epsilon]$	$\rightsquigarrow_{s}$
ϵ	yy	$y@(y@\epsilon)$	$[x_1 \leftarrow y@(y@\epsilon)] : [x_0 \leftarrow y@\epsilon]$	$\rightsquigarrow_{c_1}$
$y \Diamond y@(y@\epsilon)$	y	ϵ	$[x_1 \leftarrow y@(y@\epsilon)] : [x_0 \leftarrow y@\epsilon]$	$\rightsquigarrow_{c_3}$
ϵ	y	$(y@\epsilon) : (y@(y@\epsilon))$	$[x_1 \leftarrow y@(y@\epsilon)] : [x_0 \leftarrow y@\epsilon]$	

The initial state is the compilation of the term t_2, the final state decodes to the term $i_2 := (yy)(yy)$, and the two β-transitions in the execution correspond to the two β_i-steps in the derivation considered in Proposition 3. Each β-transition is followed by two substitution transitions (amid commutative ones): the first two substitutions replace a variable occurrence with $i_0 = y$, a term of size $2^1 - 1 = 1$; while the second two substitutions replace a variable occurrence with $i_1 = yy$, a term of size $2^2 - 1 = 3$. Note that yy is not a subterm of the initial term t_2.

Let us now consider the execution ρ_n starting on the generic term t_n in the size exploding family of Proposition 3, and explain why it is unreasonable. It turns out that the length of ρ_n is polynomial in the *size of the initial term* and in the *number of β-transitions*, so the problem is not at the level of the *number of transitions* taken by the machine. Size explosion indeed happens at the level of the *cost of single transitions*, and precisely at the level of subterms duplicated by transition $\rightsquigarrow_{s}$. As the example suggests, ρ_n at some point duplicates twice i_{n-1}, which has size $2^n - 1$. So, the cost of single substitution steps becomes exponential. Let us stress it once again: i_{n-1} is not a subterm of the initial term—the reasonable machines of the next sections avoid size explosion, and the deep reason behind this is that they only duplicate subterms of the initial term.

5. Easy GLAMOUr

In this section we introduce the *Easy GLAMOUr*, a simplified version of the GLAMOUr machine from [10] (the GLAMOUr is recalled in Sect. 9): unlike the latter, the Easy GLAMOUr does not need any labeling of codes to provide a reasonable implementation of the right-to-left evaluation $\to_{\mathtt{r}\beta_f}$ of λ_{fire}.

Background. GLAMOUr stands for *Useful* (*i.e.* optimized to be *reasonable*) *Open* (reducing also open terms) *Global* (using a single global environment) Leroy Abstract Machine (LAM). In [10] the study of the GLAMOUr was done according to the distillation approach of [12], *i.e.* by decoding the machine towards a λ-calculus with explicit substitutions. Here we do not follow the distillation approach, we decode directly to λ_{fire}, which is simpler.

Machine components. The *data structures* (items ϕ, stacks π, dumps D, global environments E, states s) used by the Easy GLAMOUr are exactly the same as the ones for the Open GLAM, formally defined in Table 2 on p. 11. We write $E(x) = \phi$ if $E = E'[x \leftarrow \phi]E''$—this functional notation for environments makes sense thanks to the invariants of forthcoming Lemma 5.1.

Transitions. The Easy GLAMOUr is defined in Table 4. Its transitions are just slight variations over those of the Open GLAM ($\rightsquigarrow_{c_1}$, $\rightsquigarrow_{c_2}$ and $\rightsquigarrow_{\beta}$ are identical), and the discussion in Sect. 4 about their functioning is still valid. In particular, garbage collection and open terms are handled in the same way, that is, there is no garbage collection and open variables are handled by condition $E(x) = \bot$ in $\rightsquigarrow_{c_3}$.

The only—crucial—difference with respect to the Open GLAM is that the Easy GLAMOUr *substitutes abstractions* but *not inert terms*, because their substitution can cause open size explosion (see Proposition 3). This difference is encapsulated in the *useful* side-conditions $E(x) = y@\pi''$ for $\rightsquigarrow_{c_3}$ and $E(x) = \lambda y.\overline{u}@\epsilon$ for $\rightsquigarrow_{s}$. Removing the useful side-conditions one

Table 4
Transitions of the Easy GLAMOUr. In the substitution transition $\rightsquigarrow_{s}$, $(\lambda y.\overline{u})^{\alpha}$ is any well-named code α-equivalent to $\lambda y.\overline{u}$ such that its bound variables are fresh with respect to those in D, π and $E[x\leftarrow\lambda y.\overline{u}@\epsilon]E'$.

Dump	Code	Stack	Global Env.		Dump	Code	Stack	Global Env.
D	$\overline{t}\,\overline{u}$	π	E	$\rightsquigarrow_{c_1}$	$D : \overline{t}\lhd\pi$	$\overline{u}$	ϵ	E
$D : \overline{t}\lhd\pi$	$\lambda x.\overline{u}$	ϵ	E	$\rightsquigarrow_{c_2}$	D	$\overline{t}$	$(\lambda x.\overline{u}@\epsilon) : \pi$	E
$D : \overline{t}\lhd\pi$	x	π'	E	$\rightsquigarrow_{c_3}$	D	$\overline{t}$	$(x@\pi') : \pi$	E
								if $E(x) = \bot$ or $E(x) = y@\pi''$
D	$\lambda x.\overline{t}$	$\phi : \pi$	E	$\rightsquigarrow_{\beta}$	D	$\overline{t}$	π	$[x\leftarrow\phi]E$
D	x	π	$E[x\leftarrow\lambda y.\overline{u}@\epsilon]E'$	$\rightsquigarrow_{s}$	D	$(\lambda y.\overline{u})^{\alpha}$	π	$E[x\leftarrow\lambda y.\overline{u}@\epsilon]E'$

recovers the Open GLAM, which is unreasonable as seen in Sect. 4. Note that the Easy GLAMOUr avoids the substitution of *all* inert terms, even when these are simple variables. As already mentioned, this fact will be relevant in the next sections.

Avoiding the substitution of inert terms. Let us provide an abstract view of why the Easy GLAMOUr is a reasonable abstract machine. We showed that *evaluation* and *substitution of inert terms* commute in λ_{fire} (Proposition 4). Consequently, substitutions of inert terms can safely be *postponed*. Now, recall that abstract machines compute compact representation of results, where *compactness* is the sharing of subterms provided by the environment. Substituting inert terms at the end would diminish the level of compactness of the result, and would reintroduce size-explosion. Rather than postpone it, then, we simply *avoid* the substitution of inert terms. It is an acceptable solution, because results have to be taken as compact anyway. Another way of seeing the postponement/avoidance of substituting inert terms, is that it is rather encapsulated in the decoding.

Compiling, decoding, and invariants. A term t is compiled to the *initial state* $t^{\circ} = (\epsilon, \overline{t}, \epsilon, \epsilon)$, where $\overline{t}$ is a well-named code α-equivalent to t: this is the starting state for an Easy GLAMOUr execution simulating the right-to-left evaluation in λ_{fire} of t. Conversely, every machine state s decodes to a term $\underline{s}$ via the same decoding function $\underline{\cdot}$ defined for the Open GLAM (see the right part of Table 2). Clearly, $\underline{t^{\circ}} = t$ for any term t, and so $(\underline{s})^{\circ} = s$ for any initial state s (as it is of the form $s = t^{\circ}$).

Example 3. To have a glimpse of how the Easy GLAMOUr works, we show how it implements the derivation $t := (\lambda z.z(yz))\lambda x.x \rightarrow^{2}_{\mathtt{r}\beta_f} y\,\lambda x.x$ of Example 1 (p. 6):

Dump	Code	Stack	Global Environment	
ϵ	$(\lambda z.z(yz))\lambda x.x$	ϵ	ϵ	$\rightsquigarrow_{c_1}$
$\lambda z.z(yz)\lhd\epsilon$	$\lambda x.x$	ϵ	ϵ	$\rightsquigarrow_{c_2}$
ϵ	$\lambda z.z(yz)$	$\lambda x.x@\epsilon$	ϵ	$\rightsquigarrow_{\beta}$
ϵ	$z(yz)$	ϵ	$[z\leftarrow\lambda x.x@\epsilon]$	$\rightsquigarrow_{c_1}$
$z\lhd\epsilon$	yz	ϵ	$[z\leftarrow\lambda x.x@\epsilon]$	$\rightsquigarrow_{c_1}$
$z\lhd\epsilon : y\lhd\epsilon$	z	ϵ	$[z\leftarrow\lambda x.x@\epsilon]$	$\rightsquigarrow_{s}$
$z\lhd\epsilon : y\lhd\epsilon$	$\lambda x'.x'$	ϵ	$[z\leftarrow\lambda x.x@\epsilon]$	$\rightsquigarrow_{c_2}$
$z\lhd\epsilon$	y	$\lambda x'.x'@\epsilon$	$[z\leftarrow\lambda x.x@\epsilon]$	$\rightsquigarrow_{c_3}$
ϵ	z	$y@(\lambda x'.x'@\epsilon)$	$[z\leftarrow\lambda x.x@\epsilon]$	$\rightsquigarrow_{s}$
ϵ	$\lambda x''.x''$	$y@(\lambda x'.x'@\epsilon)$	$[z\leftarrow\lambda x.x@\epsilon]$	$\rightsquigarrow_{\beta}$
ϵ	x''	ϵ	$[x''\leftarrow y@(\lambda x'.x'@\epsilon)] : [z\leftarrow\lambda x.x@\epsilon]$	

Note that the initial state is the compilation of the term t, the final state decodes to the term $y\,\lambda x.x$, and the two β-transitions in the execution correspond to the two $\mathtt{r}\beta_f$-steps in the derivation considered in Example 1.

Example 4. Let us see how the Easy GLAMOUr implements the size-exploding derivation $t_2 := (\lambda x_1.x_1x_1)t_1 = (\lambda x_1.x_1x_1)((\lambda x_0.x_0x_0)y) \rightarrow^{2}_{\beta_i} (yy)(yy) =: i_2$ of Proposition 3 (for $n = 2$), which we also considered for the Open GLAM in Example 2:

Dump	Code	Stack	Global Environment	
ϵ	$(\lambda x_1.x_1x_1)t_1$	ϵ	ϵ	$\rightsquigarrow_{c_1}$
$\lambda x_1.x_1x_1\lhd\epsilon$	$(\lambda x_0.x_0x_0)y$	ϵ	ϵ	$\rightsquigarrow_{c_1}$
$\lambda x_1.x_1x_1\lhd\epsilon : \lambda x_0.x_0x_0\lhd\epsilon$	y	ϵ	ϵ	$\rightsquigarrow_{c_3}$
$\lambda x_1.x_1x_1\lhd\epsilon$	$\lambda x_0.x_0x_0$	$y@\epsilon$	ϵ	$\rightsquigarrow_{\beta}$
$\lambda x_1.x_1x_1\lhd\epsilon$	x_0x_0	ϵ	$[x_0\leftarrow y@\epsilon]$	$\rightsquigarrow_{c_1}$
$\lambda x_1.x_1x_1\lhd\epsilon : x_0\lhd\epsilon$	x_0	ϵ	$[x_0\leftarrow y@\epsilon]$	$\rightsquigarrow_{c_3}$
$\lambda x_1.x_1x_1\lhd\epsilon$	x_0	$x_0@\epsilon$	$[x_0\leftarrow y@\epsilon]$	$\rightsquigarrow_{c_3}$
ϵ	$\lambda x_1.x_1x_1$	$x_0@(x_0@\epsilon)$	$[x_0\leftarrow y@\epsilon]$	$\rightsquigarrow_{\beta}$
ϵ	x_1x_1	ϵ	$[x_1\leftarrow x_0@(x_0@\epsilon)] : [x_0\leftarrow y@\epsilon]$	$\rightsquigarrow_{c_1}$
$x_1\lhd\epsilon$	x_1	ϵ	$[x_1\leftarrow x_0@(x_0@\epsilon)] : [x_0\leftarrow y@\epsilon]$	$\rightsquigarrow_{c_3}$
ϵ	x_1	$x_1@\epsilon$	$[x_1\leftarrow x_0@(x_0@\epsilon)] : [x_0\leftarrow y@\epsilon]$	

Note that the initial state is the compilation of the term t_2, the final state decodes to the term $i_2 := (yy)(yy)$, and the two β-transitions in the execution correspond to the two β_i-steps in the derivation considered in Proposition 3.

Comparing with the execution of the Open GLAM implementing the same derivation (Example 2), the two executions have different final states, and yet they decode to the same term—the Easy GLAMOUr produces more compact final states. Since the Easy GLAMOUr does not substitute inert terms, the final state in the Easy GLAMOUr is reached without performing any substitution transition. This is how the size explosion problem that affects the Open GLAM is circumvented.

The study of the Easy GLAMOUr machine relies on the following invariants.

Lemma 5 *(Easy GLAMOUr qualitative invariants).*[6] *Let $s = (D, \bar{t}, \pi, E)$ be a reachable state of an Easy GLAMOUr execution. Then:*

1. Name*:*
   (a) Explicit substitution*: if $E = E'[x \leftarrow \phi]E''$ then the variable x is fresh with respect to ϕ and E'';*
   (b) Abstraction*: if $\lambda x.\overline{u}$ is a subterm of $D, \bar{t}, \pi$ or E, then the variable x may occur only in $\overline{u}$.*
2. Fireball item*: for every item ϕ in π, in E or in any stack in D, one has that $\underline{\phi}$ and $\underline{\phi}\downarrow_E$ are inert terms if $\phi = x@\pi'$, and abstractions otherwise.*
3. Contextual decoding*: $R_s = \underline{D}\langle\underline{\pi}\rangle\downarrow_E$ is a right context.*

Proof. Easy induction on the length of the execution ending in the state s. □

The fireball item invariant (Lemma 5.2) entails that there is no reachable state whose environment has the form $E = E_0[y \leftarrow x@\pi]E'[x \leftarrow \lambda z.\bar{t}@\epsilon]E''$, because otherwise the unfolding $\underline{x@\pi}\downarrow_E = \langle\lambda z.(\bar{t}\downarrow_E)\rangle(\underline{\pi}\downarrow_E)$ of the item $x@\pi$ in E would not be an inert term. Moreover, by the name invariant for explicit substitutions (Lemma 5.1a), the environment of any reachable state must be of the form $[x_1 \leftarrow \phi_1]\dots[x_n \leftarrow \phi_n]$ where $n \geq 0$ and variables $x_1, \dots, x_n$ are pairwise distinct.

Implementation theorem. The invariants above are used to prove the implementation theorem for the Easy GLAMOUr (Theorem 2 below) by showing that the hypotheses of Theorem 1 hold, namely that the Easy GLAMOUr forms an implementation system with respect to right-to-left evaluation $\to_{\mathtt{r}\beta_f}$ in the fireball calculus, via the decoding $\underline{\cdot}$. More precisely, we employ the invariants to prove the next lemma, which states that the Easy GLAMOUr transitions project on the fireball calculus λ_{fire} (*i.e.* Points 1-2 of Definition 2 are fulfilled). Note that substitution ($\rightsquigarrow_{\mathtt{s}}$) and commutative ($\rightsquigarrow_{\mathtt{c}_{1,2,3}}$) transitions are considered as overhead transitions.

Lemma 6 *(Easy GLAMOUr β-projection and overhead transparency). Let s be a reachable state of an Easy GLAMOUr execution.*

1. β-projection*: if $s \rightsquigarrow_\beta s'$ then $\underline{s} \to_{\mathtt{r}\beta_f} \underline{s'}$;*
2. Overhead transparency*: if $s \rightsquigarrow_{\mathtt{s},\mathtt{c}_{1,2,3}} s'$ then $\underline{s} = \underline{s'}$.*[7]

Proof. Let us check all the transitions, listed according to their order in Table 4:

1. $s = (D, \bar{t}\overline{u}, \pi, E) \rightsquigarrow_{\mathtt{c}_1} (D : \bar{t} \Diamond \pi, \overline{u}, \epsilon, E) = s'$. Then
$$\underline{s} = \underline{D}\langle\langle\bar{t}\overline{u}\rangle\underline{\pi}\rangle\downarrow_E = \underline{D : \bar{t} \Diamond \pi}\langle\overline{u}\rangle\downarrow_E = \underline{D : \bar{t} \Diamond \pi}\langle\langle\overline{u}\rangle\underline{\epsilon}\rangle\downarrow_E = \underline{s'}.$$
2. $s = (D : \bar{t} \Diamond \pi, \lambda x.\overline{u}, \epsilon, E) \rightsquigarrow_{\mathtt{c}_2} (D, \bar{t}, (\lambda x.\overline{u}@\epsilon) : \pi, E) = s'$. Then
$$\underline{s} = \underline{D : \bar{t} \Diamond \pi}\langle\langle\lambda x.\overline{u}\rangle\underline{\epsilon}\rangle\downarrow_E = \underline{D}\langle\langle\bar{t}(\langle\lambda x.\overline{u}\rangle\underline{\epsilon})\rangle\underline{\pi}\rangle\downarrow_E = \underline{D}\langle\langle\bar{t}\rangle\underline{(\lambda x.\overline{u}@\epsilon) : \pi}\rangle\downarrow_E = \underline{s'}.$$
3. $s = (D : \bar{t} \Diamond \pi, x, \pi', E) \rightsquigarrow_{\mathtt{c}_3} (D, \bar{t}, (x@\pi') : \pi, E) = s'$ where $E(x) = \bot$ or $E(x) = y@\pi''$. Then
$$\underline{s} = \underline{D : \bar{t} \Diamond \pi}\langle\langle x\rangle\underline{\pi'}\rangle\downarrow_E = \underline{D}\langle\langle\bar{t}(\langle x\rangle\underline{\pi'})\rangle\underline{\pi}\rangle\downarrow_E = \underline{D}\langle\langle\bar{t}\rangle\underline{(x@\pi') : \pi}\rangle\downarrow_E = \underline{s'}.$$
4. $s = (D, \lambda x.\bar{t}, \phi : \pi, E) \rightsquigarrow_\beta (D, \bar{t}, \pi, [x \leftarrow \phi]E) = s'$. Then
$$\underline{s} = \underline{D}\langle\langle\lambda x.\bar{t}\rangle\underline{\phi : \pi}\rangle\downarrow_E = \underline{D}\langle\langle(\lambda x.\bar{t})\underline{\phi}\rangle\underline{\pi}\rangle\downarrow_E \to_{\mathtt{r}\beta_f} \underline{D}\langle\langle\bar{t}\{x \leftarrow \underline{\phi}\}\rangle\underline{\pi}\rangle\downarrow_E = \underline{D}\langle\langle\bar{t}\rangle\underline{\pi}\rangle\{x \leftarrow \underline{\phi}\}\downarrow_E = \underline{D}\langle\langle\bar{t}\rangle\underline{\pi}\rangle\downarrow_{[x \leftarrow \phi]E} = \underline{s'}$$
where the $\mathtt{r}\beta_f$-step takes place because $\underline{\phi}$ is a fireball by the fireball item invariant (Lemma 5.2), and $\underline{D}\langle\underline{\pi}\rangle\downarrow_E$ is a right context by the contextual decoding invariant (Lemma 5.3). Moreover, the meta-level substitution $\{x \leftarrow \underline{\phi}\}$ can be extruded (in the equality step after $\to_{\mathtt{r}\beta_f}$) without renaming x, as x does not occur in D or π by the name invariant (Lemma 5.1b).

[6] Proof in Appendix, p. 28.

[7] Given two transitions $\rightsquigarrow_{\mathtt{r}_1}$ and $\rightsquigarrow_{\mathtt{r}_2}$ for a machine $\mathtt{M}$, we set $\rightsquigarrow_{\mathtt{r}_1,\mathtt{r}_2} := \rightsquigarrow_{\mathtt{r}_1} \cup \rightsquigarrow_{\mathtt{r}_2}$ (also denoted by $\rightsquigarrow_{\mathtt{r}_{1,2}}$ or simply $\rightsquigarrow_{\mathtt{r}}$). Thus, in this case $\rightsquigarrow_{\mathtt{s},\mathtt{c}_{1,2,3}} = \rightsquigarrow_{\mathtt{s}} \cup \rightsquigarrow_{\mathtt{c}_1} \cup \rightsquigarrow_{\mathtt{c}_2} \cup \rightsquigarrow_{\mathtt{c}_3}$.

5. $s = (D, x, \pi, E) \rightsquigarrow_{\mathrm{s}} (D, (\lambda y.\overline{u})^{\alpha}, \pi, E) = s'$ where $E = E_1[x \leftarrow \lambda y.\overline{u}@\epsilon]E_2$. Then, by Lemma 5.1a,

$$\underline{s} = \underline{D}\langle\langle x\rangle\underline{\pi}\rangle\downarrow_E = \underline{D}\downarrow_E\langle\langle x\downarrow_E\rangle\underline{\pi}\downarrow_E\rangle = \underline{D}\downarrow_E\langle\langle\lambda y.\underline{\overline{u}}\downarrow_E\rangle\underline{\pi}\downarrow_E\rangle = \underline{D}\langle\langle\lambda y.\overline{u}\rangle\underline{\pi}\rangle\downarrow_E = \underline{s'}. \quad \square$$

We also need a lemma about the halt condition for an implementation system (Point 5 of Definition 2).

Lemma 7 *(Easy GLAMOUr halt). Let s be a reachable final state of an Easy GLAMOUr execution. Then $\underline{s}$ is a fireball, i.e. it is β_f-normal.*

Proof. An immediate inspection of the transitions shows that in a final state the code cannot be an application and the dump is necessarily empty. In fact, a final state s has one of the following two shapes:

1. *Top-level unapplied abstraction, i.e.* $s = (\epsilon, \lambda x.\overline{t}, \epsilon, E)$. Then $\underline{s} = (\lambda x.\underline{\overline{t}})\downarrow_E = \lambda x.(\underline{\overline{t}}\downarrow_E)$, which is an abstraction and so a fireball.
2. *Top-level inert term with free head, i.e.* $s = (\epsilon, x, \pi, E)$ with $E(x) = \bot$ or $E(x) = y@\pi'$. Subcases:
   (a) $E(x) = \bot$. Then $\underline{s} = (\langle x\rangle\underline{\pi})\downarrow_E = \langle x\downarrow_E\rangle(\underline{\pi}\downarrow_E) = \langle x\rangle(\underline{\pi}\downarrow_E)$ is inert because $\underline{\phi}\downarrow_E$ is a fireball for every item ϕ in π, according to the fireball item invariant (Lemma 5.2);
   (b) $E(x) = y@\pi'$. Then $\underline{s} = (\langle x\rangle\underline{\pi})\downarrow_E = \langle x\downarrow_E\rangle(\underline{\pi}\downarrow_E) = \langle\underline{y@\pi'}\downarrow_E\rangle(\underline{\pi}\downarrow_E)$ is inert because $\underline{\phi}\downarrow_E$ is a fireball for every item ϕ in π and $\underline{y@\pi'}\downarrow_E$ is inert, according to the fireball item invariant (Lemma 5.2). $\square$

Finally, we obtain the implementation theorem for the Easy GLAMOUr.

Theorem 2 *(Easy GLAMOUr implementation). The Easy GLAMOUr implements right-to-left evaluation $\rightarrow_{\mathrm{r}\beta_f}$ in λ_{fire} (via the decoding $\underline{\cdot}$).*

Proof. According to Theorem 1, it is enough to show that the Easy GLAMOUr, the right-to-left evaluation $\rightarrow_{\mathrm{r}\beta_f}$ and the decoding $\underline{\cdot}$ form an implementation system, *i.e.* that the five conditions in Definition 2 hold. By Lemma 6, Points 1-2 (β-projection and overhead transparency) of Definition 2 are fulfilled. By Lemma 7, also Point 5 (halt) of Definition 2 holds, since $\rightarrow_{\mathrm{r}\beta_f} \subseteq \rightarrow_{\beta_f}$. Let us prove Points 3-4 of Definition 2.

3. *Overhead termination*: termination of transitions $\rightsquigarrow_{\mathrm{s},\mathrm{c}_{1,2,3}}$ is given by the forthcoming Lemma 9 and Lemma 10, which are postponed because they actually give precise complexity bounds, not just termination.
4. *Determinism*: the Easy GLAMOUr machine is deterministic, as it can be seen by an easy inspection of the transitions (see Table 4). Lemma 1.2 proves that $\rightarrow_{\mathrm{r}\beta_f}$ is deterministic. $\square$

6. Complexity analysis of the Easy GLAMOUr

The analysis of the Easy GLAMOUr is done according to the recipe given at the end of Sect. 3. The result (see Theorem 3 below) is that the Easy GLAMOUr is linear in the number $|\rho|_\beta$ of β-steps/transitions and quadratic in the size $|t_0|$ of the initial term t_0, *i.e.* the overhead of implementing an execution ρ on a RAM has complexity $O((1 + |\rho|_\beta) \cdot |t_0|^2)$.

The analysis relies on a quantitative invariant, the crucial *subterm invariant*, ensuring that $\rightsquigarrow_{\mathrm{s}}$ duplicates only subterms of the initial term, so that the cost of duplications is connected to one of the two parameters for complexity analyses. The statement of the subterm invariant is about abstractions, not duplications, but note that abstractions are the only terms duplicated by the Easy GLAMOUr, according to the substitution transition $\rightsquigarrow_{\mathrm{s}}$ in Table 4.

Lemma 8 *(Subterm invariant). Let $\rho : t_0^{\circ} \rightsquigarrow^* (D, \overline{t}, \pi, E)$ be an Easy GLAMOUr execution. Every subterm $\lambda x.\overline{u}$ of D, $\overline{t}$, π or E is a subterm of t_0.*

Proof. First, a clarification about *subterms*: for us, $\overline{u}$ is a subterm of t_0 if it is one up to variable names, both free and bound (the distinction between term and code is then irrelevant). More precisely: let t^- be t in which all variables (including those appearing in binders) are replaced by a fixed symbol $*$. So, we define u as a subterm of t whenever u^- is a subterm of t^- in the usual sense. The key property ensured by this definition is that the size $|\overline{u}|$ of $\overline{u}$ is bounded by $|\overline{t}|$.

Now, the proof is by induction on the length $|\rho|$ of the execution ρ. In the initial state the invariant trivially holds. For a non-empty execution the proof is by a simple case analysis on the last transition, always relying on the *i.h.* $\square$

Remark. The unreasonable Open GLAM seen in Sect. 4 does satisfy the subterm invariant as it is stated above (*i.e.* about abstractions), but it duplicates also inert terms (indeed, in the transition $\rightsquigarrow_{\mathrm{s}}$ in Table 3, ϕ can decode to an abstraction or an inert term), and not all inert terms in its reachable states are subterms of the initial term. In this sense, as already pointed out in Example 2, the Open GLAM does not satisfy the subterm invariant—morally, this is what makes it unreasonable.

Intuition about complexity bounds. Given an execution $\rho : t_0^\circ \rightsquigarrow^* s$, the number $|\rho|_{\mathrm{s}}$ of substitution transitions $\rightsquigarrow_{\mathrm{s}}$ in ρ depends on both parameters for complexity analyses, the number $|\rho|_\beta$ of β-transitions in ρ *and* the size $|t_0|$ of the initial term. Dependence on $|\rho|_\beta$ is standard, and appears in every machine [17,33,12,10,13,2]—sometimes it is quadratic, here it is linear, in Sect. 7 we come back to this point. Dependence on $|t_0|$ is also always present, but usually only for *the cost* of a single $\rightsquigarrow_{\mathrm{s}}$ transition, since only subterms of t_0 are duplicated, as ensured by the subterm invariant (Lemma 8). For the Easy GLAMOUr, instead, also *the number* $|\rho|_{\mathrm{s}}$ of $\rightsquigarrow_{\mathrm{s}}$ transitions depends on $|t_0|$: this is a side-effect of dealing with open terms. Since both the cost and the number of $\rightsquigarrow_{\mathrm{s}}$ transitions depend linearly on $|t_0|$, the overall contribution of $\rightsquigarrow_{\mathrm{s}}$ transitions to the overhead in an implementation of ρ on RAM depends quadratically on $|t_0|$.

The family of terms below shows the quadratic dependence on $|t_0|$ in isolation (*i.e.*, with no dependence on $|\rho|_\beta$). Let $r_n := \lambda x.(\dots((y\underbrace{x)x)\dots)x}_{n}$ and consider:

$$u_n := r_n r_n = (\lambda x.(\dots((y\overbrace{x)x)\dots)x}^{n})r_n \to_{\beta_\lambda} (\dots((y\overbrace{r_n)r_n)\dots)r_n}^{n}. \tag{4}$$

Forgetting about commutative transitions, the Easy GLAMOUr would evaluate u_n with one β-transition $\rightsquigarrow_\beta$ and n substitution transitions $\rightsquigarrow_{\mathrm{s}}$, each one duplicating r_n, whose size (as well as the size of the initial term u_n) is linear in n.

The number $|\rho|_{\mathrm{c}}$ of commutative transitions $\rightsquigarrow_{\mathrm{c}}$, roughly, is linear in the amount of code involved in the execution ρ. More precisely, one has to count the code out of abstractions or that is exposed when a β-transition removes an abstraction and then the machine evaluates its body. This amount is bounded by the initial code plus the code exposed by β-transitions, which in turn is bounded by the number of β-transitions times the size of the initial term. The number of commutative transitions is then $O((1 + |\rho|_\beta) \cdot |t_0|)$. As each one has constant cost, this is also a bound on their overall cost in an implementation of ρ on RAM.

Number of Transitions 1: substitution vs. β transitions. The number $|\rho|_{\mathrm{s}}$ of substitution transitions in an execution $\rho : t_0^\circ \rightsquigarrow^* s$ is proven (see Lemma 9 below) to be bilinear, *i.e.* linear in $|t_0|$ and $|\rho|_\beta$, by means of a measure $|\cdot|_{\mathsf{free}}$ always bounded by the size of the code, *i.e.* such that $0 \le |\bar{t}|_{\mathsf{free}} \le |\bar{t}|$ for any code $\bar{t}$. The *free size* $|\cdot|_{\mathsf{free}}$ of a code counts the number of (free) variable occurrences that are not under abstractions. It is defined and extended to states as follows:

$$|x|_{\mathsf{free}} := 1 \qquad |\lambda y.\overline{u}|_{\mathsf{free}} := 0 \qquad |\bar{t}\overline{u}|_{\mathsf{free}} := |\bar{t}|_{\mathsf{free}} + |\overline{u}|_{\mathsf{free}} \qquad |(D, \bar{t}, \pi, E)|_{\mathsf{free}} := |\bar{t}|_{\mathsf{free}} + \textstyle\sum_{\overline{u}\diamond\pi' \in D} |\overline{u}|_{\mathsf{free}}.$$

Lemma 9 *(Bilinear number of substitution transitions). Let $\rho : t_0^\circ \rightsquigarrow^* s$ be an Easy GLAMOUr execution. Then, $|\rho|_{\mathrm{s}} \le |\rho|_{\mathrm{s}} + |s|_{\mathsf{free}} \le (1 + |\rho|_\beta) \cdot |t_0|$.*

Proof. First, $|\rho|_{\mathrm{s}} \le |\rho|_{\mathrm{s}} + |s|_{\mathsf{free}}$ because $|s|_{\mathsf{free}} \ge 0$. We prove that $|\rho|_{\mathrm{s}} + |s|_{\mathsf{free}} \le (1 + |\rho|_\beta) \cdot |t_0|$ by induction on $|\rho|$.

If $|\rho| = 0$ then $t_0^\circ = s$ and $|\rho|_\beta = 0 = |\rho|_{\mathrm{s}}$, so the statement becomes $|\overline{t_0}|_{\mathsf{free}} \le |\overline{t_0}| = |t_0|$ that is always true.

Otherwise $\sigma : t_0^\circ \rightsquigarrow^* s'$ and ρ extends σ with $s' \rightsquigarrow s$. By *i.h.*, $|\sigma|_{\mathrm{s}} + |s'|_{\mathsf{free}} \le (1 + |\sigma|_\beta) \cdot |t_0|$. Cases (the notation refers to the transitions of the machine, in Table 4):

- *the last transition is a substitution transition.* The inequality $|\rho|_{\mathrm{s}} + |s|_{\mathsf{free}} \le (1 + |\rho|_\beta) \cdot |t_0|$ follows from the *i.h.* and the fact that $|\rho|_{\mathrm{s}} = |\sigma|_{\mathrm{s}} + 1$, $|\rho|_\beta = |\sigma|_\beta$ and $|s|_{\mathsf{free}} = |s'|_{\mathsf{free}} - 1$, where the last equality holds because the dump does not change and the code changes from a variable (of measure 1) to an abstraction (of measure 0);
- *the last transition is a β-transition.* Then:

$$\begin{aligned}
|\rho|_{\mathrm{s}} + |s|_{\mathsf{free}} &= |\rho|_{\mathrm{s}} + |\bar{t}|_{\mathsf{free}} + \textstyle\sum_{\overline{u}\diamond\pi' \in D} |\overline{u}|_{\mathsf{free}} \\
&= |\rho|_{\mathrm{s}} + |\bar{t}|_{\mathsf{free}} + |\lambda x.\bar{t}|_{\mathsf{free}} + \textstyle\sum_{\overline{u}\diamond\pi' \in D} |\overline{u}|_{\mathsf{free}} && (|\lambda x.\bar{t}|_{\mathsf{free}} = 0) \\
&= |\rho|_{\mathrm{s}} + |s'|_{\mathsf{free}} + |\bar{t}|_{\mathsf{free}} && (\text{def. of } |s'|_{\mathsf{free}}) \\
&\le |\rho|_{\mathrm{s}} + |s'|_{\mathsf{free}} + |\bar{t}| && (|\bar{t}|_{\mathsf{free}} \le |\bar{t}|) \\
&\le |\rho|_{\mathrm{s}} + |s'|_{\mathsf{free}} + |t_0| && (\text{Lemma 8}) \\
&= |\sigma|_{\mathrm{s}} + |s'|_{\mathsf{free}} + |t_0| && (|\rho|_{\mathrm{s}} = |\sigma|_{\mathrm{s}}) \\
&\le (1 + |\sigma|_\beta) \cdot |t_0| + |t_0| && (\textit{i.h.}) \\
&= |\rho|_\beta \cdot |t_0| + |t_0| = (1 + |\rho|_\beta) \cdot |t_0| && (|\rho|_\beta = |\sigma|_\beta + 1);
\end{aligned}$$

- *the last transition is a commutative transition.* The inequality $|\rho|_{\mathrm{s}} + |s|_{\mathsf{free}} \le (1 + |\rho|_\beta) \cdot |t_0|$ follows from the *i.h.* and the fact that $|\rho|_{\mathrm{s}} = |\sigma|_{\mathrm{s}}$, $|\rho|_\beta = |\sigma|_\beta$ and either $|s|_{\mathsf{free}} = |s'|_{\mathsf{free}}$ (when the last transitions is $\rightsquigarrow_{\mathrm{c}_1}$ or $\rightsquigarrow_{\mathrm{c}_2}$) or $|s|_{\mathsf{free}} = |s'|_{\mathsf{free}} - 1$ (when the last transition is $\rightsquigarrow_{\mathrm{c}_3}$). □

Number of transitions 2: commutative vs. β transitions. The bound on the number $|\rho|_{\mathrm{c}}$ of commutative transitions in an execution $\rho : t_0^\circ \rightsquigarrow^* s$ is obtained by means of a (different) measure $|\cdot|_{\mathrm{c}}$ of codes and states. The bound is bilinear in $|\rho|_\beta$ and $|t_0|$: this analysis improves the one in [8], where the bound was linear in $|\rho|_\beta$ and quadratic in $|t_0|$.

The *commutative size* $|\cdot|_{\mathrm{c}}$ of codes and states is defined as follows:

$$|x|_{\mathrm{c}} := 1 \qquad |\lambda y.\overline{u}|_{\mathrm{c}} := 1 \qquad |\overline{t}\,\overline{u}|_{\mathrm{c}} := |\overline{t}|_{\mathrm{c}} + |\overline{u}|_{\mathrm{c}} + 1 \qquad |(D, \overline{t}, \pi, E)|_{\mathrm{c}} := |\overline{t}|_{\mathrm{c}} + \textstyle\sum_{\overline{u}\Diamond\pi' \in D} |\overline{u}|_{\mathrm{c}}\,.$$

Note that $1 \leq |\overline{t}|_{\mathrm{c}} \leq |\overline{t}|$ for every code $\overline{t}$. As in the case of $|s|_{\mathsf{free}}$, the definition of $|s|_{\mathrm{c}}$ for a state s takes into account only the current code and the left code of each item in the dump.

Lemma 10 *(Bilinear number of commutative transitions). Let $\rho : t_0^\circ \rightsquigarrow^* s$ be an Easy GLAMOUr execution. Then, $|\rho|_{\mathrm{c}} \leq |\rho|_{\mathrm{c}} + |s|_{\mathrm{c}} \leq (1 + |\rho|_\beta) \cdot |t_0|$.*

Proof. First, note that $|\rho|_{\mathrm{c}} \leq |\rho|_{\mathrm{c}} + |s|_{\mathrm{c}}$ since $|s|_{\mathrm{c}} \geq 0$. We prove that $|\rho|_{\mathrm{c}} + |s|_{\mathrm{c}} \leq (1 + |\rho|_\beta) \cdot |t_0|$ by induction on the length $|\rho|$ of the execution ρ.

Base case (empty execution, *i.e.* $|\rho| = 0$): $t_0^\circ = s$ and $|\rho|_{\mathrm{c}} = 0 = |\rho|_\beta$, so the property collapses on the tautology $|\overline{t}_0| \leq |t_0|$.

Inductive case ($|\rho| > 0$): let $s' \rightsquigarrow s$ be the last transition of ρ and let σ be the prefix of ρ ending on s'. By the *i.h.* applied to σ, $|\sigma|_{\mathrm{c}} + |s'|_{\mathrm{c}} \leq (1 + |\sigma|_\beta) \cdot |t_0|$. We now prove that the statement holds by analyzing the various cases of $s' \rightsquigarrow s$ and showing that the inequality holds also after the transition:

- *Commutative transitions* $\rightsquigarrow_{\mathrm{c}_1}$: the rule splits the code $\overline{t}\,\overline{u}$ (of size $|\overline{t}|_{\mathrm{c}} + |\overline{u}|_{\mathrm{c}} + 1$) between the dump and the code. Thus $|s|_{\mathrm{c}} = |s'|_{\mathrm{c}} - 1$, while $|\rho|_{\mathrm{c}} = |\sigma|_{\mathrm{c}} + 1$ and $|\rho|_\beta = |\sigma|_\beta$. So, $|\rho|_{\mathrm{c}} + |s|_{\mathrm{c}} = |\sigma|_{\mathrm{c}} + |s'|_{\mathrm{c}} \leq (1 + |\sigma|_\beta) \cdot |t_0| = (1 + |\rho|_\beta) \cdot |t_0|$.
- *Commutative transitions* $\rightsquigarrow_{\mathrm{c}_{2,3}}$: these rules consume the current code that is a variable or an abstraction (of commutative size 1), so $|s|_{\mathrm{c}} = |s'|_{\mathrm{c}} - 1$. Since $|\rho|_{\mathrm{c}} = |\sigma|_{\mathrm{c}} + 1$ and $|\rho|_\beta = |\sigma|_\beta$, it follows that—as in the previous case—neither the lhs nor the rhs changes, hence the inequality is preserved.
- *β-transition* $\rightsquigarrow_\beta$: it modifies the current code by replacing an abstraction $\lambda x.\overline{t}$ (of commutative size 1) with its body $\overline{t}$ (of commutative size ≥ 1). Then, $|s|_{\mathrm{c}} = |s'|_{\mathrm{c}} + |\overline{t}|_{\mathrm{c}} - 1 \leq |s'|_{\mathrm{c}} + |t|$. By the subterm invariant (Lemma 8), t is a subterm of t_0, hence $|t| \leq |t_0|$ and so $|s|_{\mathrm{c}} \leq |s'|_{\mathrm{c}} + |t_0|$. Since $|\rho|_{\mathrm{c}} = |\sigma|_{\mathrm{c}}$ and $|\rho|_\beta = |\sigma|_\beta + 1$, one has $|\rho|_{\mathrm{c}} + |s|_{\mathrm{c}} \leq |\sigma|_{\mathrm{c}} + |s'|_{\mathrm{c}} + |t_0| \leq (1 + |\sigma|_\beta) \cdot |t_0| + |t_0| = (1 + |\rho|_\beta) \cdot |t_0|$.
- *Substitution transition* $\rightsquigarrow_{\mathrm{s}}$: in the current code, it replaces a variable with an abstraction (both of commutative size 1), so $|s|_{\mathrm{c}} = |s'|_{\mathrm{c}}$ and neither the lhs nor the rhs changes (as $|\rho|_{\mathrm{c}} = |\sigma|_{\mathrm{c}}$ and $|\rho|_\beta = |\sigma|_\beta$). Thus, the inequality is preserved. □

Cost of single transitions. We need to make some hypotheses on how the Easy GLAMOUr is going to be itself implemented on a random access machine (RAM):

1. *Codes, stacks, variable (occurrences) and environment entries*: abstractions and applications are constructors with pointers to subterms; similarly, a pair $\overline{t}@\pi$ or $\overline{t}\Diamond\pi$ is a constructor with pointers to the code $\overline{t}$ and the stack π. A stack is a singly linked list. A variable is a memory location, a variable occurrence is a reference to it, and an environment entry $[x \leftarrow \phi]$ is the fact that the location associated with x contains (the topmost constructor of) ϕ.
2. *Random access to global environments*: the environment E can be accessed in $O(1)$ (in the transition $\rightsquigarrow_{\mathrm{s}}$) by just following the reference given by the variable occurrence under evaluation, with no need to access E sequentially, thus ignoring its list structure (used only to ease the definition of decoding).

With these hypotheses it is clear that each β and commutative transition can be implemented in $O(1)$, since it only accesses the head of a list and redirects some pointers. Each substitution transition needs to copy a code from the environment (the renaming $\overline{t}^{\alpha}$) and can be implemented in $O(|t_0|)$, since the code to copy is a subterm of the input t_0 by the subterm invariant (Lemma 8) and the environment can be accessed in $O(1)$.

These hypotheses are *realistic*, as they are verified for instance by some of the OCaml implementations of abstract machines in Accattoli and Barras [5].

Summing up. By putting together the bounds on the number of transitions with the cost of single transitions we obtain the complexity of the Easy GLAMOUr, *i.e.* the overhead when it is implemented on RAM.

Theorem 3 *(Easy GLAMOUr overhead bound). Let $\rho : t_0^\circ \rightsquigarrow^* s$ be an Easy GLAMOUr execution. Then ρ is implementable on RAM in $O((1 + |\rho|_\beta) \cdot |t_0|^2)$, i.e. linear in the number $|\rho|_\beta$ of β-transitions (aka the length of the derivation $d : t_0 \to^*_{\mathrm{r}\beta_f} \underline{s}$ implemented by ρ) and quadratic in the size of the initial term t_0.*

Proof. The cost of implementing ρ is the sum of the costs of implementing its β, substitution and commutative transitions on RAM:

Table 5
Transitions of the Fast GLAMOUr. In the transition $\rightsquigarrow_{\mathrm{s}}$, $(\lambda y.\overline{u})^{\alpha}$ is any well-named code α-equivalent to $\lambda y.\overline{u}$ such that its bound variables are fresh with respect to those in D, π and $E[x\leftarrow\lambda y.\overline{u}@\epsilon]E'$. Data-structures, unfolding and decoding are defined as in Table 2.

Dump	Code	Stack	Global Env.		Dump	Code	Stack	Global Env.
D	$\overline{t}\,\overline{u}$	π	E	$\rightsquigarrow_{\mathrm{c}_1}$	$D : \overline{t}\Diamond\pi$	$\overline{u}$	ϵ	E
$D : \overline{t}\Diamond\pi$	$\lambda x.\overline{u}$	ϵ	E	$\rightsquigarrow_{\mathrm{c}_2}$	D	$\overline{t}$	$(\lambda x.\overline{u}@\epsilon) : \pi$	E
$D : \overline{t}\Diamond\pi$	x	π'	E	$\rightsquigarrow_{\mathrm{c}_3}$	D	$\overline{t}$	$(x@\pi') : \pi$	E
					if $E(x) = \bot$ or $E(x) = y@\pi''$ or $(E(x) = \lambda y.\overline{u}@\epsilon$ and $\pi' = \epsilon)$			
D	$\lambda x.\overline{t}$	$(y@\epsilon) : \pi$	E	$\rightsquigarrow_{\beta_1}$	D	$\overline{t}\{x\leftarrow y\}$	π	E
D	$\lambda x.\overline{t}$	$\phi : \pi$	E	$\rightsquigarrow_{\beta_2}$	D	$\overline{t}$	π	$[x\leftarrow\phi]E$
								if $\phi \neq y@\epsilon$
D	x	$\phi : \pi$	$E[x\leftarrow\lambda y.\overline{u}@\epsilon]E'$	$\rightsquigarrow_{\mathrm{s}}$	D	$(\lambda y.\overline{u})^{\alpha}$	$\phi : \pi$	$E[x\leftarrow\lambda y.\overline{u}@\epsilon]E'$

1. *β-transition* $\rightsquigarrow_{\beta}$: each one costs $O(1)$ and so all together they cost $O(|\rho|_{\beta})$.
2. *Substitution transition* $\rightsquigarrow_{\mathrm{s}}$: by Lemma 9 we have $|\rho|_{\mathrm{s}} \leq (1 + |\rho|_{\beta}) \cdot |t_0|$, *i.e.* the number of substitution transitions is bilinear in $|\rho|_{\beta}$ and $|t_0|$. By the subterm invariant (Lemma 8), each substitution transition costs at most $O(|t_0|)$, and so their full cost is $O((1 + |\rho|_{\beta}) \cdot |t_0|^2)$.
3. *Commutative transitions* $\rightsquigarrow_{\mathrm{c}}$: by Lemma 10 we have $|\rho|_{\mathrm{c}} \leq (1 + |\rho|_{\beta}) \cdot |t_0|$. Since each commutative transition evidently takes constant time, the whole cost of the commutative transitions is bounded by $O((1 + |\rho|_{\beta}) \cdot |t_0|)$. □

7. Fast GLAMOUr

In this section we optimize the Easy GLAMOUr, obtaining a machine, the *Fast GLAMOUr*, whose dependence on the size of the initial term is linear, instead of quadratic, providing a bilinear—thus optimal—overhead (see Theorem 5 below and compare it with Theorem 3 on the Easy GLAMOUr). We invite the reader to go back to derivation (4) on page 16, where the quadratic dependence was explained: in that example the substitutions of r_n do not create β_f-redexes, and so they are useless. The Fast GLAMOUr avoids these useless substitutions and it implements the example with no substitutions at all.

Optimization: abstractions on-demand. The difference between the Easy GLAMOUr and the machines in [10] (to be surveyed in Sect. 9) is that, whenever the former encounters a variable occurrence x bound to an abstraction $\lambda y.\overline{t}$ in the environment, it replaces x with $\lambda y.\overline{t}$, while the latter are more parsimonious. They implement an optimization that we call *substituting abstractions on-demand*: x is replaced by $\lambda y.\overline{t}$ only if this is useful to obtain a β-redex, that is, only if the argument stack is non-empty. The Fast GLAMOUr, defined in Table 5, upgrades the Easy GLAMOUr with *substitutions of abstractions on-demand*—note the new side-condition for $\rightsquigarrow_{\mathrm{c}_3}$ and the non-empty stack in $\rightsquigarrow_{\mathrm{s}}$.

Abstractions on-demand and the substitution of variables. The new optimization however has a consequence. To explain it, let us recall the role of another optimization, *no substitution of variables*. In the Easy GLAMOUr, abstractions are at depth 1 in the environment: there cannot be chains of renamings—*i.e.* of substitutions of variables for variables—ending in abstractions (such as $[x\leftarrow y@\epsilon][y\leftarrow z@\epsilon][z\leftarrow\lambda z'.\overline{t}@\epsilon]$). This property implies that the overhead is linear in $|\rho|_{\beta}$ and it is induced by the fact that variables cannot be substituted (for variables). If variables can be substituted then the overhead becomes quadratic in $|\rho|_{\beta}$—this is what happens in the GLAMOUr machine in [10] (see Sect. 9). The relationship between *substituting variables* and a linear/quadratic overhead is studied in-depth by Accattoli and Sacerdoti Coen [11].

Now, because the Fast GLAMOUr substitutes abstractions on-demand, variable occurrences that are not applied are not substituted by abstractions. The question becomes what to do when the code is an abstraction $\lambda x.\overline{t}$ and the top of the stack argument ϕ is a simple variable occurrence $\phi = y@\epsilon$ (potentially bound to an abstraction in the environment E) because if one admits that $[x\leftarrow y@\epsilon]$ is added to E then the depth of abstractions in the environment may be arbitrary and so the dependence on $|\rho|_{\beta}$ may be quadratic, as in the GLAMOUr.

There are two possible solutions to this issue. The complex one, given by the Unchaining GLAMOUr in [10], is to add labels and a further unchaining optimization (that is overviewed in Sect. 9). The simple one explored here is to resort to *compacting β-transitions*. The technique consists in splitting the β-transition in two, handling this situation with a new rule that renames x as y in the code $\overline{t}$ without touching the environment—this is exactly what the Fast GLAMOUr does with $\rightsquigarrow_{\beta_1}$ and $\rightsquigarrow_{\beta_2}$. The consequence is that abstractions stay at depth 1 in E, and so the overhead is indeed bilinear.

The simple solution is taken from Sands, Gustavsson, and Moran [33], where they use it on a call-by-name machine. Actually, it repeatedly appears in the literature on abstract machines even well before [33], often with reference to space consumption and *space leaks*, for instance in Wand [36], Friedman et al. [24], and Sestoft [34]. Sands, Gustavsson, and Moran were however the first to show that compacting βs have an effect on the *time* complexity of executions.

Fast GLAMOUr. The machine is in Table 5—note the two kinds of β-transitions. Data-structures, compilation, and decoding are as for the Easy GLAMOUr.

Example 5. Let us now show how the derivation $t := (\lambda z.z(yz))\lambda x.x \to^2_{\mathtt{r}\beta_f} y\,\lambda x.x$ of Example 1 is implemented by the Fast GLAMOUr. The execution is similar to that of the Easy GLAMOUr in Example 3, since they implement the same derivation and hence have the same initial state. In particular, the first five transitions in the Fast GLAMOUr (omitted here) are the same as in the Easy GLAMOUr (see Example 3 and replace $\rightsquigarrow_\beta$ with $\rightsquigarrow_{\beta_2}$). Then, the Fast GLAMOUr executes:

Dump	Code	Stack	Global Environment	
$z\langle\rangle\epsilon : y\langle\rangle\epsilon$	z	ϵ	$[z\leftarrow\lambda x.x@\epsilon]$	$\rightsquigarrow_{\mathtt{c}_3}$
$z\langle\rangle\epsilon$	y	$z@\epsilon$	$[z\leftarrow\lambda x.x@\epsilon]$	$\rightsquigarrow_{\mathtt{c}_3}$
ϵ	z	$y@(z@\epsilon)$	$[z\leftarrow\lambda x.x@\epsilon]$	$\rightsquigarrow_{\mathtt{s}}$
ϵ	$\lambda x''.x''$	$y@(z@\epsilon)$	$[z\leftarrow\lambda x.x@\epsilon]$	$\rightsquigarrow_{\beta_2}$
ϵ	x''	ϵ	$[x''\leftarrow y@(z@\epsilon)] : [z\leftarrow\lambda x.x@\epsilon]$	

The Fast GLAMOUr executes only one substitution transition (the Easy GLAMOUr takes two) since the replacement of z with $\lambda x.x$ from the environment is *on-demand* (*i.e.* useful to obtain a β-redex) only for the first occurrence of z in $z(yz)$.

For the Fast GLAMOUr we proceed like for the Easy GLAMOUr in Sect. 5: we use invariants to prove that the Fast GLAMOUr implements right-to-left evaluation $\to_{\mathtt{r}\beta_f}$ of the fireball calculus (via the decoding $\underline{\cdot}$). The differences are minimal with respect to statements and proofs for the Easy GLAMOUr (detailed proofs are in Appendix B.3 for the sake of completeness). In particular, the Fast GLAMOUr satisfies the same qualitative invariants as the Easy GLAMOUr (Lemma 5) except for the *fireball item*, which is slightly different.

Lemma 11 *(Fast GLAMOUr qualitative invariants).*[8] *Let $s = (D, \overline{t}, \pi, E)$ be a reachable state of a Fast GLAMOUr execution. Then:*

1. Name:
   (a) Explicit substitution*: if $E = E'[x\leftarrow\phi]E''$ then the variable x is fresh with respect to ϕ and E'';*
   (b) Abstraction*: if $\lambda x.\overline{u}$ is a subterm of D, $\overline{t}$, π or E, then the variable x may occur only in $\overline{u}$.*
2. Fireball item*: for every item ϕ in π, in E or in any stack in D, one has that $\underline{\phi}$ and $\underline{\phi}\downarrow_E$ are: inert terms if $\phi = x@\pi'$ and either $E(x) = \bot$ or $E(x) = y@\pi''$; abstractions otherwise.*
3. Contextual decoding*: $R_s = \underline{D}\langle\underline{\pi}\rangle\downarrow_E$ is a right context.*

Proof. Easy induction on the length of the execution ending in the state s. □

Analogously to the Easy GLAMOUr, the invariants above are used to prove the implementation theorem for the Fast GLAMOUr (Theorem 4) by showing that the hypotheses of Theorem 1 hold, namely that the Fast GLAMOUr forms an implementation system (Definition 2) with respect to right-to-left evaluation $\to_{\mathtt{r}\beta_f}$ in the fireball calculus λ_{fire}, via the decoding $\underline{\cdot}$.

First, we show (Lemma 12) that the two conditions about the projection of Fast GLAMOUr transitions on the fireball calculus (*i.e.* Points 1-2 of Definition 2) are fulfilled. Its proof is analogous to the corresponding lemma for the Easy GLAMOUr (Lemma 6). Substitution ($\rightsquigarrow_{\mathtt{s}}$) and commutative ($\rightsquigarrow_{\mathtt{c}_{1,2,3}}$) transitions are considered as overhead transitions, while the β transitions are $\rightsquigarrow_{\beta_1}$ and $\rightsquigarrow_{\beta_2}$.

Lemma 12 *(Fast GLAMOUr β-projection and overhead transparency).*[9] *Let s be a reachable state of a Fast GLAMOUr execution.*

1. Overhead Transparency*: if $s \rightsquigarrow_{\mathtt{s},\mathtt{c}_{1,2,3}} s'$ then $\underline{s} = \underline{s'}$;*
2. β-Projection*: if $s \rightsquigarrow_{\beta_{1,2}} s'$ then $\underline{s} \to_{\mathtt{r}\beta_f} \underline{s'}$.*

The next lemma deals with the halt condition for an implementation system (Point 5 of Definition 2). It is proved similarly to the corresponding lemma for the Easy GLAMOUr (Lemma 7).

Lemma 13 *(Fast GLAMOUr halt).*[10] *Let s be a reachable final state of a Fast GLAMOUr execution. Then $\underline{s}$ is a fireball, i.e. it is β_f-normal.*

We can now prove the implementation theorem for the Fast GLAMOUr.

Theorem 4 *(Fast GLAMOUr implementation). The Fast GLAMOUr implements right-to-left evaluation $\to_{\mathtt{r}\beta_f}$ in λ_{fire} (via the decoding $\underline{\cdot}$).*

[8] Proof in Appendix, p. 30.
[9] Proof in Appendix, p. 31.
[10] Proof in Appendix, p. 31.

Proof. According to Theorem 1, it is enough to show that the Fast GLAMOUr, the right-to-left evaluation $\to_{\mathfrak{r}\beta_f}$ and the decoding $\underline{\cdot}$ form an implementation system, *i.e.* that the five conditions in Definition 2 hold. By Lemma 12, Points 1-2 (β-projection and overhead transparency) of Definition 2 are fulfilled. By Lemma 13, also Point 5 (halt) of Definition 2 holds, since $\to_{\mathfrak{r}\beta_f} \subseteq \to_{\beta_f}$. Let us prove Points 3-4 of Definition 2.

3. *Overhead termination*: Termination of transitions $\rightsquigarrow_{\mathrm{s},\mathrm{c}_{1,2,3}}$ is given by the forthcoming Lemma 14, which is postponed because it actually gives precise complexity bounds, not just termination.
4. *Determinism*: The Fast GLAMOUr machine is deterministic, as can be seen by an easy inspection of the transitions (see Table 5). Lemma 1.2 proves that $\to_{\mathfrak{r}\beta_f}$ is deterministic. □

Complexity analysis. What changes with respect to the Easy GLAMOUr is the complexity analysis which, surprisingly, is simpler. The Fast GLAMOUr has the same *subterm invariant* as the Easy GLAMOUr (Lemma 8, just replace "Easy GLAMOUr" with "Fast GLAMOUr" in the statement), the proof is analogous.

We focus on *the number* of overhead transitions of a Fast GLAMOUr execution ρ, expressed as a function of the number $|\rho|_\beta$ of β-transitions and the size $|t_0|$ of the initial term t_0 (Lemma 14). The *substitution vs. β transitions* part is simply trivial: since abstractions are substituted only on-demand, it is easy to show that the number $|\rho|_\mathrm{s}$ of substitution transitions is simply bounded by $|\rho|_\beta$, without any dependence on $|t_0|$ (differently from the Easy GLAMOUr, see Lemma 9).

For the *commutative vs. β transitions* part, the same measure and the same reasoning of the Easy GLAMOUr seen in Lemma 10 provide the same bound on the number $|\rho|_\mathrm{c}$ of commutative transitions, namely $|\rho|_\mathrm{c} \leq (1 + |\rho|_\beta) \cdot |t_0|$.

Lemma 14 *(Number of overhead transitions). Let $\rho : t_0^\circ \rightsquigarrow^* s$ be a Fast GLAMOUr execution. Then,*

1. Substitution *vs.* β transitions*:* $|\rho|_\mathrm{s} \leq |\rho|_\beta$.
2. Commutative *vs.* β transitions*:* $|\rho|_\mathrm{c} \leq (1 + |\rho|_\beta) \cdot |t_0|$.

Proof. 1. *Substitution vs. β transitions*: since abstractions are substituted only on-demand, every substitution transition that is not the last transition is immediately followed by a β-transition. Therefore, in an execution ρ there can be at most one substitution transition not followed by a β-transition, and so $|\rho|_\mathrm{s} \leq |\rho|_\beta + 1$. The $+1$ can be easily removed: ρ must have a β_2 transition before the first substitution one, otherwise the environment is empty and no substitution is possible; thus $|\rho|_\mathrm{s} \leq |\rho|_\beta$.

2. *Commutative vs. β transitions*: the bound $|\rho|_\mathrm{c} \leq (1 + |\rho|_\beta) \cdot |t_0|$, which is the same as in the Easy GLAMOUr, is obtained in exactly the same way as in the proof of Lemma 10, by using the commutative size $|\cdot|_\mathrm{c}$ for states defined in Sect. 6. The differences in the proof are minimal:

   - Transitions $\rightsquigarrow_{\mathrm{c}_1}$ and $\rightsquigarrow_{\mathrm{c}_2}$: no difference, because they are exactly the same transitions as in the Easy GLAMOUr.
   - Transition $\rightsquigarrow_{\mathrm{c}_3}$: the transition has a side-condition more than the corresponding transition in the Easy GLAMOUr. Then it is a sub-case, and so the bound obviously hold.
   - Transition $\rightsquigarrow_{\beta_1}$: the novelty of the transition is the renaming of the code, but it leaves the commutative size, and thus the inequality, unchanged.
   - Transition $\rightsquigarrow_{\beta_2}$: a special case of $\rightsquigarrow_\beta$ in the Easy GLAMOUr.
   - Transition $\rightsquigarrow_\mathrm{s}$: a special case of $\rightsquigarrow_\mathrm{s}$ in the Easy GLAMOUr. □

Cost of single transitions and global overhead. For the cost of single transitions, note that $\rightsquigarrow_{\mathrm{c}_1}$, $\rightsquigarrow_{\mathrm{c}_2}$, $\rightsquigarrow_{\mathrm{c}_3}$ and $\rightsquigarrow_{\beta_2}$ have (evidently) cost $O(1)$ while $\rightsquigarrow_\mathrm{s}$ and $\rightsquigarrow_{\beta_1}$ have cost $O(|t_0|)$ by the subterm invariant. Therefore, we can conclude with:

Theorem 5 *(Fast GLAMOUr bilinear overhead). Let $\rho : t_0^\circ \rightsquigarrow^* s$ be a Fast GLAMOUr execution. Then ρ is implementable on RAM in $O((1 + |\rho|_\beta) \cdot |t_0|)$, i.e. linear in both the number $|\rho|_\beta$ of β-transitions (aka the length of the derivation $d : t_0 \to^*_{\mathfrak{r}\beta_f} \underline{s}$ implemented by ρ) and the size of the initial term t_0.*

Proof. The cost of implementing ρ is the sum of the costs of implementing its β, substitution and commutative transitions on RAM:

1. *β-transitions $\rightsquigarrow_{\beta_1}$ and $\rightsquigarrow_{\beta_2}$*: each transition $\rightsquigarrow_{\beta_1}$ costs $O(|t_0|)$ because the code has to be renamed and by the subterm invariant (Lemma 8 for the Fast GLAMOUr) the size of the code is bounded by $|t_0|$. Each transition $\rightsquigarrow_{\beta_2}$ instead takes constant time. In the worst case, β-transitions all together cost $O(|\rho|_\beta \cdot |t_0|)$.
2. *Substitution transition $\rightsquigarrow_\mathrm{s}$*: by Lemma 14, we have $|\rho|_\mathrm{s} \leq |\rho|_\beta$. By the subterm invariant (Lemma 8 for the Fast GLAMOUr), each substitution transition costs at most $O(|t_0|)$, and so their full cost is $O(|\rho|_\beta \cdot |t_0|)$.
3. *Commutative transitions $\rightsquigarrow_\mathrm{c}$*: by Lemma 14, $|\rho|_\mathrm{c} \leq (1 + |\rho|_\beta) \cdot |t_0|$. Since every commutative transition evidently takes constant time, the whole cost of the commutative transitions is bounded by $O((1 + |\rho|_\beta) \cdot |t_0|)$. □

Table 6
Transitions of the BE GLAMOUr. The variable y in the right-hand side of β transition $\rightsquigarrow_\beta$ is fresh with respect to the variables in D, ϕ, π, and E.

Dump	Code	Stack	Global Env.		Dump	Code	Stack	Global Env.
D	$\bar{t}\bar{u}$	π	E	$\rightsquigarrow_{c_1}$	$D : \bar{t}\Diamond\pi$	$\bar{u}$	ϵ	E
$D : \bar{t}\Diamond\pi$	$\lambda x.\bar{u}$	ϵ	E	$\rightsquigarrow_{c_2}$	D	$\bar{t}$	$(\lambda x.\bar{u}@\epsilon) : \pi$	E
$D : \bar{t}\Diamond\pi$	x	π'	E	$\rightsquigarrow_{c_3}$	D	$\bar{t}$	$(x@\pi') : \pi$	E
							if $E(x) = \bot$ or $E(x) = y@\pi''$	
D	$\lambda x.\bar{t}$	$\phi : \pi$	E	$\rightsquigarrow_\beta$	D	$\bar{t}\{x \leftarrow y\}$	π	$[y \leftarrow \phi]E$
D	x	π	$E[x \leftarrow \lambda y.\bar{u}@\epsilon]E'$	$\rightsquigarrow_s$	D	$\lambda y.\bar{u}$	π	$E[x \leftarrow \lambda y.\bar{u}@\epsilon]E'$

Table 7
Transitions of the BEST GLAMOUr. The variable y in the right-hand side of β transition $\rightsquigarrow_{\beta_2}$ is fresh with respect to the variables in D, ϕ, π, and E.

Dump	Code	Stack	Global Env.		Dump	Code	Stack	Global Env.
D	$\bar{t}\bar{u}$	π	E	$\rightsquigarrow_{c_1}$	$D : \bar{t}\Diamond\pi$	$\bar{u}$	ϵ	E
$D : \bar{t}\Diamond\pi$	$\lambda x.\bar{u}$	ϵ	E	$\rightsquigarrow_{c_2}$	D	$\bar{t}$	$(\lambda x.\bar{u}@\epsilon) : \pi$	E
$D : \bar{t}\Diamond\pi$	x	π'	E	$\rightsquigarrow_{c_3}$	D	$\bar{t}$	$(x@\pi') : \pi$	E
							if $E(x) = \bot$ or $E(x) = y@\pi''$ or $(E(x) = \lambda y.\bar{u}@\epsilon$ and $\pi' = \epsilon)$	
D	$\lambda x.\bar{t}$	$(y@\epsilon) : \pi$	E	$\rightsquigarrow_{\beta_1}$	D	$\bar{t}\{x \leftarrow y\}$	π	E
D	$\lambda x.\bar{t}$	$\phi : \pi$	E	$\rightsquigarrow_{\beta_2}$	D	$\bar{t}\{x \leftarrow y\}$	π	$[y \leftarrow \phi]E$
								if $\phi \neq y@\epsilon$
D	x	$\phi : \pi$	$E[x \leftarrow \lambda y.\bar{u}@\epsilon]E'$	$\rightsquigarrow_s$	D	$\lambda y.\bar{u}$	$\phi : \pi$	$E[x \leftarrow \lambda y.\bar{u}@\epsilon]E'$

8. A further refinement: renaming on β

In this section we sketch the *renaming on* β optimization, that moves part of the complexity of the substitution transition to the β-transition. Its effect on the Easy GLAMOUr is relevant, as it makes its overhead bilinear, while on the Fast GLAMOUr it is somewhat negligible, as it provides a small improvement but it does not change the overall complexity of the machine.

Renaming on β is already at work in *e.g.* Sands et al. [33], and Danvy and Zerny [22] where however it has no impact on the complexity of the machines, as they deal with closed calculi. We discovered the relevance of this optimization in the open setting after our paper was submitted, in a joint work with Condoluci and Sacerdoti Coen [14], building on the present one, which is why we present it as an additional optimization.

Let us point out that *renaming on* β does not turn *substituting abstractions on-demand*—the optimization of the Fast GLAMOUr—into a useless optimization: in Sect. 11 we show that substituting abstractions on-demand is *mandatory* for a reasonable implementation of Strong CbV.

The BE GLAMOUr. Table 6 presents the *BE GLAMOUr*, a minor variant of the Easy GLAMOUr (BE comes from *renaming on Beta Easy*). The only difference is that the α-renaming now happens on the β-transition $\rightsquigarrow_\beta$ and no longer on the substitution transition $\rightsquigarrow_s$. It is the body $\bar{t}$ of the abstraction in the code that is α-renamed, and—to avoid name clashes—it is enough to rename only the abstracted variable x with a fresh one y, without α-renaming all the bound variables in the body $\bar{t}$.

The machine has the same subterm invariant as the Easy GLAMOUr, which is why the change is harmless: the code $\bar{t}$ of a reachable state is a subterm of the initial term t_0, whose size then bounds the cost of the new renaming operation $\bar{t}\{x \leftarrow y\}$, exactly as in the case of α-renaming on the substitution transition.

Moving the renaming operation has the following effects on the complexity analysis, for an execution ρ of initial term t_0:

1. *β-transition* $\rightsquigarrow_\beta$: now each one costs $|t_0|$ by the subterm invariant, so their global cost is $O(|\rho|_\beta \cdot |t_0|)$, that is higher than in the Easy GLAMOUr and yet bilinear;
2. *Substitution transition* $\rightsquigarrow_s$: the number $|\rho|_s$ of substitution transitions is unchanged with respect to the Easy GLAMOUr ($|\rho|_s \leq (1 + |\rho|_\beta) \cdot |t_0|$), *i.e.* it is bilinear in $|\rho|_\beta$ and $|t_0|$. The cost of each such transition however now is *constant*, because the copy of the code from the environment can be done by simply copying a pointer, since renaming is no longer needed. Therefore, the global cost of substitutions lowers to $O((1 + |\rho|_\beta) \cdot |t_0|)$.
3. *Commutative transitions* $\rightsquigarrow_{c_{1,2,3}}$: nothing changes, the reasoning for the Easy GLAMOUr still applies, so the number $|\rho|_c$ of commutative transitions is bilinear in $|\rho|_\beta$ and $|t_0|$, and the cost of each such transition is constant. Therefore, their global cost is bilinear, *i.e.* $O((1 + |\rho|_\beta) \cdot |t_0|$.

The complexity of the BE GLAMOUr is then *bilinear*, *i.e.* $O((1 + |\rho|_\beta) \cdot |t_0|)$.

The BEST GLAMOUr. Renaming on β can also be applied to the Fast GLAMOUr, obtaining the *BEST GLAMOUr* (renaming on BEta faST GLAMOUr), whose transitions are in Table 7. The optimization lowers the global cost of substitution transitions to $O(|\rho|_\beta)$ (since $|\rho|_s \leq |\rho|_\beta$ as in the Fast GLAMOUr, but there is no α-renaming) while that one of β-transitions becomes $O(|\rho|_\beta \cdot |t_0|)$ (for the same reason as for the BE GLAMOUr). On the whole, the complexity of the BEST GLAMOUr is still

Table 8
Transitions of the GLAMOUr. In the substitution transition $\rightsquigarrow_{s}$, $(\underline{\psi})^{\alpha}$ is any well-named code α-equivalent to $\underline{\psi}$ such that its bound variables are fresh with respect to those in D, $\phi^{\ell}:\pi$ and $E[x\leftarrow\psi^{\lambda}]E'$.

Dump	Code	Stack	Global Env.		Dump	Code	Stack	Global Env.
D	$\overline{t}\,\overline{u}$	π	E	$\rightsquigarrow_{c_1}$	$D:\overline{t}\Diamond\pi$	$\overline{u}$	ϵ	E
$D:\overline{t}\Diamond\pi$	$\lambda x.\overline{u}$	ϵ	E	$\rightsquigarrow_{c_2}$	D	$\overline{t}$	$(\lambda x.\overline{u}@\epsilon)^{\lambda}:\pi$	E
$D:\overline{t}\Diamond\pi$	x	π'	E	$\rightsquigarrow_{c_3}$	D	$\overline{t}$	$(x@\pi')^{i}:\pi$	E
							if $E(x)=\bot$ or	$E(x)=(y@\pi'')^{i}$
$D:\overline{t}\Diamond\pi$	x	ϵ	$E[x\leftarrow\phi^{\lambda}]E'$	$\rightsquigarrow_{c_4}$	D	$\overline{t}$	$(x@\epsilon)^{\lambda}:\pi$	$E[x\leftarrow\phi^{\lambda}]E'$
D	$\lambda x.\overline{t}$	$\phi^{\ell}:\pi$	E	$\rightsquigarrow_{\beta}$	D	$\overline{t}$	π	$[x\leftarrow\phi^{\ell}]E$
D	x	$\phi^{\ell}:\pi$	$E[x\leftarrow\psi^{\lambda}]E'$	$\rightsquigarrow_{s}$	D	$(\underline{\psi})^{\alpha}$	$\phi^{\ell}:\pi$	$E[x\leftarrow\psi^{\lambda}]E'$

bilinear, more precisely $O((1+|\rho|_{\beta})\cdot|t_0|)$ (as in the Fast/BE GLAMOUr), because of commutative transitions that are identical to the Fast GLAMOUr and hence have the same global cost. The difference between the Fast and the BEST GLAMOUr is minimal, renaming on β however renames only one variable and only when needed, while renaming on substitutions renames all the variables in the body of the abstraction and in an eager way, and so the BEST GLAMOUr is slightly better.

9. GLAMOUr and Unchaining GLAMOUr

In this section we quickly recall the *GLAMOUr* and *Unchaining GLAMOUr* from Accattoli and Sacerdoti Coen [10], to explain the differences with respect to the Easy GLAMOUr and the Fast GLAMOUr. We are somewhat going to provide an alternative explanation to the issues that have been discussed at the beginning of Sect. 7, to introduce the Fast GLAMOUr.

9.1. GLAMOUr

The machine is in Table 8. The basic ideas are that the GLAMOUr:

1. *never substitutes inert terms that are not variables*, and so it is reasonable;
2. *substitutes variables* (for variables), so its overhead is *quadratic* in the number $|\rho|_{\beta}$ of β transitions, as outlined on p. 18;
3. *substitutes abstractions on demand*, and so its overhead is *linear* in the size $|t_0|$ of the initial term;
4. relies on *labels* (explained below), to implement Points 1 and 3; labels are needed because of Point 2.

The complexity of the GLAMOUr is then $O((1+|\rho|_{\beta}^{2})\cdot|t_0|)$.

Labels. The key point is understanding how the labeling works. Every stack item ϕ (as usual, stack items are also the bodies of environment entries) has a label $\ell\in\{\lambda,i\}$ indicating whether its unfolded decoding $\underline{\phi\downarrow_{E}}$ relative to the global environment E (see the definitions in Table 2) is an abstraction (case $\ell=\lambda$) or an inert term (case $\ell=i$). This property is of course guaranteed to be one of the invariants of the machine (for reachable states).

Labels are needed because environment entries can contain variables, *i.e.* stack items of the form $\phi=y@\epsilon$. Let us first focus on the need for labels, and postpone for a moment the explanation of why environment entries contain variables.

The need for labels. The issue comes from the fact that not every variable has to be replaced by what is in the environment, because inert terms shall not be substituted and abstraction shall be substituted only on demand (*i.e.* when the stack is non-empty—note the stack $\phi^{\ell}:\pi$ in transition $\rightsquigarrow_{s}$). When the current code is a variable x, looking at the entry $[x\leftarrow\phi]$ (if any) in the environment E is not enough to know whether x will end up being replaced by an inert term or by an abstraction, because ϕ may be a variable, and so to retrieve the needed information one needs to keep exploring the environment, potentially following a chain of variables until an abstraction or an inert term is found. Labels allow us to keep this information local, circumventing the exploration of the environment.

Why environment entries contain variables. It is a side effect of substituting abstractions on demand. Note transition $\rightsquigarrow_{c_4}$: when the variable x to be substituted by (something that will become) an abstraction is not applied (*i.e.* the stack is empty) and so the substitution is not needed on that occurrence of x, the machine backtracks and puts $(x@\epsilon)^{\lambda}$ on the stack, which might then be pushed onto the environment by a subsequent β transition. If, instead, abstractions are substituted whenever, the problem disappears, and one obtains the Easy GLAMOUr of Sect. 5; the complexity of the overhead however changes: the dependence on the initial term then becomes quadratic.

9.2. Unchaining GLAMOUr

The Unchaining GLAMOUr machine is in Table 9. It is exactly as the GLAMOUr but for the substitution rule $\rightsquigarrow_{s}$, that is replaced by a more sophisticated *unchaining mechanism* given by the new three rules $\rightsquigarrow_{s_1}$, $\rightsquigarrow_{c_5}$, and $\rightsquigarrow_{s_2}$. These rules dynamically shorten the chains of variables entries in the environment E. A new data structure is also required, the *heap* H,

Table 9
Transitions of the Unchaining GLAMOUr. In the substitution transition $\rightsquigarrow_{s_1}$, $(\lambda x.\overline{u})^{\alpha}$ is any well-named code α-equivalent to $\lambda x.\overline{u}$ such that its bound variables are fresh with respect to those in D, $\phi^{\ell}:\pi$ and $E[x\leftarrow(\lambda x.\overline{u}@\epsilon)^{\lambda}]E'$. In the substitution transition $\rightsquigarrow_{s_2}$, $E^{\bullet} = E'[y\leftarrow(x@\epsilon)^{\lambda}]E''[x\leftarrow(\lambda z.\overline{u}@\epsilon)^{\lambda}]E'''$ and $E^{\circ} = E'[y\leftarrow((\lambda z.\overline{u})^{\alpha}@\epsilon)^{\lambda}]E''[x\leftarrow(\lambda z.\overline{u}@\epsilon)^{\lambda}]E'''$ where $(\lambda z.\overline{u})^{\alpha}$ is any code α-equivalent to $\lambda z.\overline{u}$ that preserves the well-naming of the machine.

Dump	Heap	Code	Stack	Global Env.		Dump	Heap	Code	Stack	Global Env.
D	ϵ	$\overline{t}\overline{u}$	π	E	$\rightsquigarrow_{c_1}$	$D:\overline{t}\Diamond\pi$	ϵ	$\overline{u}$	ϵ	E
$D:\overline{t}\Diamond\pi$	ϵ	$\lambda x.\overline{u}$	ϵ	E	$\rightsquigarrow_{c_2}$	D	ϵ	$\overline{t}$	$(\lambda x.\overline{u}@\epsilon)^{\lambda}:\pi$	E
$D:\overline{t}\Diamond\pi$	ϵ	x	π'	E	$\rightsquigarrow_{c_3}$	D	ϵ	$\overline{t}$	$(x@\pi')^{i}:\pi$	E
									if $E(x)=\bot$ or	$E(x)=(y@\pi'')^{i}$
$D:\overline{t}\Diamond\pi$	ϵ	x	ϵ	$E[x\leftarrow\phi^{\lambda}]E'$	$\rightsquigarrow_{c_4}$	D	ϵ	$\overline{t}$	$(x@\epsilon)^{\lambda}:\pi$	$E[x\leftarrow\phi^{\lambda}]E'$
D	ϵ	$\lambda x.\overline{t}$	$\phi^{\ell}:\pi$	E	$\rightsquigarrow_{\beta}$	D	ϵ	$\overline{t}$	π	$[x\leftarrow\phi^{\ell}]E$
D	ϵ	x	$\phi^{\ell}:\pi$	$E[x\leftarrow(\lambda z.\overline{u}@\epsilon)^{\lambda}]E'$	$\rightsquigarrow_{s_1}$	D	ϵ	$(\lambda z.\overline{u})^{\alpha}$	$\phi^{\ell}:\pi$	$E[x\leftarrow(\lambda z.\overline{u}@\epsilon)^{\lambda}]E'$
D	$H:y$	x	$\phi^{\ell}:\pi$	$E[x\leftarrow(y@\epsilon)^{\lambda}]E'$	$\rightsquigarrow_{c_5}$	D	$H:x$	y	$\phi^{\ell}:\pi$	$E[x\leftarrow(y@\epsilon)^{\lambda}]E'$
D	$H:y$	x	$\phi^{\ell}:\pi$	$E^{\bullet}$	$\rightsquigarrow_{s_2}$	D	H	y	$\phi^{\ell}:\pi$	E°

to keep track of the already visited part of the chain of variable replacements under exploration, and then properly backtrack once an abstraction entry has been found. Compared to the GLAMOUr, the unchaining mechanism lowers the overhead to be *linear* also in the number $|\rho|_{\beta}$ of β-transitions. So, the complexity of the Unchaining GLAMOUr is $O((1+|\rho|_{\beta})\cdot|t_0|)$, the same as the Fast GLAMOUr.

Eager and lazy handling of the chains. Let us postpone for a moment the precise explanation of the transitions and rather provide an abstract view of the unchaining mechanism. The difference between the Fast and the Unchaining GLAMOUr is that the former handles variable replacement chains *eagerly* (see transition $\rightsquigarrow_{\beta_1}$ in Table 5) while the latter does it *lazily*.

The compacting mechanism of the Fast GLAMOUr, indeed, avoids altogether creating variable replacement chains, by performing a substitution whenever the argument is a variable. Relative inefficiency is due to substitutions that go through the current code even when the variable to rename in fact never occur.

The unchaining mechanism of the Unchaining GLAMOUr, instead, allows for the creation of chains, but the first time that a chain has to be followed the chain itself is unchained, guaranteeing that it will not be followed again. So chains are compacted *by need*. An easy amortized analysis (in [10], not here) shows that the unchaining work is reasonable.

One of the contributions of our work is to show that the compacting approach of the Fast GLAMOUr is in fact much easier to define and analyze than the unchaining approach. Complexity-wise, however, they are equivalent.

Unchaining variable chains. Let us explain how the unchaining mechanism works. Suppose that the current code is a variable x, that the stack is non empty, and that the environment contains an entry $[x\leftarrow\phi^{\lambda}]$, so that the substitution of an abstraction on-demand has to be done. Thus the machine follows the chain of variable replacements in the environment E starting from x, until it finds an abstraction item $(\lambda z.\overline{u}@\epsilon)^{\lambda}$. Then it backtracks and simultaneously unchains the chain, by substituting (a copy of) $\lambda z.\overline{u}$ on every entry of the chain, and finally also on x, as required.

Let us have a look at the transitions in Table 9. Suppose the environment associates x with a variable item $(y@\epsilon)^{\lambda}$. Then,

1. *Chain exploration*: the environment is explored, by following the chain of variable replacements starting from x, via transition $\rightsquigarrow_{c_5}$. Concretely, x is pushed on the heap H, and the machine jumps evaluating the item, by making y the current code. The machine keeps repeating this process until it finds a variable bound to an abstraction item.
2. *Unchaining while substituting and backtracking*: the current variable x is associated with an abstraction item $(\lambda z.\overline{u}@\epsilon)^{\lambda}$ and now the backtracking starts. Two cases:

   (a) *The heap is non-empty*: this is were the unchaining happen, via transition $\rightsquigarrow_{s_2}$. Given a heap $H:y$, the environment has the form $E^{\bullet} = E'[y\leftarrow(x@\epsilon)^{\lambda}]E''[x\leftarrow(\lambda z.\overline{u}@\epsilon)^{\lambda}]E'''$. The machine substitutes the abstraction on $[y\leftarrow(x@\epsilon)^{\lambda}]$, producing the environment $E^{\circ} = E'[y\leftarrow((\lambda z.\overline{u})^{\alpha}@\epsilon)^{\lambda}]E''[x\leftarrow(\lambda z.\overline{u}@\epsilon)^{\lambda}]E'''$ and unchaining x from y. The machines also pops y from the heap, to keep backtracking the chain exploration previously done by $\rightsquigarrow_{c_5}$.
   (b) *The heap is empty*: this is the base case, that may also happen if x from the start was associated with an abstraction entry, and not a variable one. Then the current code x is simply replaced by an appropriate renaming $(\lambda z.\overline{u})^{\alpha}$ of $(\lambda z.\overline{u}@\epsilon)$, via transition $\rightsquigarrow_{s_1}$, as in the other machines. Nothing happens to the environment.

10. On the minimality of the cost model

In the previous sections we had a close look at the overhead of various machines implementing the fireball calculus λ_{fire}. These machines have all been measured with respect to the same cost model, that is, the number of fireball steps $\rightarrow_{\beta_f}$.

Here we change focus, trying to understand whether it is possible to go one step further, and switch to a more parsimonious and still natural cost model for Open CbV. The idea—inspired by the fact that evaluation and substitution of inert terms commute (Proposition 4)—is to ignore steps substituting inert terms ($\rightarrow_{\beta_i}$) and only count steps substituting abstractions ($\rightarrow_{\beta_\lambda}$). We do not have a definite answer, but we provide evidence that such a change is far from being straightforward.

Do inert steps cost 1 or 0? The polynomial complexity of GLAMOUr machines shows that the *number of fireball steps* $\to_{\beta_f}$ is a reasonable *time cost model* for Open CbV. This roughly means that the cost of an *inert step* $\to_{\beta_i}$ can be taken as 1, even if in Open CbV inert steps may cause size explosion (Proposition 3). Concretely, in a reasonable implementation of Open CbV (such as all the machines in the GLAMOUr family presented here and in [10]) this is obtained by never substituting inert terms that are not variables, thus handling a β_i-step (at least when the argument of the redex is a non-variable inert term) in constant time. It is then natural to wonder whether the cost of an inert step can actually be taken as 0, or if they are in fact computationally relevant for complexity analyses. Said differently, can inert steps be seen as administrative work akin to commutative steps? Is their cost dominated by the number of *abstraction steps* $\to_{\beta_\lambda}$?

Here we provide evidence that the cost of an inert step is 1, not 0, *i.e.* it is *relevant* and cannot be considered as administrative work, despite its constant cost. Namely, we show a *inert length exploding family*, that is, a family of terms that evaluates in a linear number of β_λ-steps followed by an *exponential* number of β_i-steps. Therefore, abstraction steps do not dominate inert steps, and it seems that the number of β_λ-steps is not a reasonable cost model for Open CbV.

Beware: we do not prove that one cannot count 0 the cost of a β_i-step, as in principle there might be an evaluation algorithm avoiding the potential exponential number of inert steps. Nonetheless, our family shows that such an algorithm, if any, is non-trivial and has to rely on some new insight to manage polynomially the exponential blow-up of inert redexes.

Inert length explosion. We build our family of terms in two steps, first identifying a family $\{u_n\}_{n\in\mathbb{N}}$ that evaluates in $\Omega(2^n)$ β_i-steps to normal form, and then building a family $\{s_n\}_{n\in\mathbb{N}}$ where each s_n evaluates in $O(n)$ β_λ-steps to u_n.

Step 1: exponentially many β_i-steps. Let i be an inert term. Consider the following three families of terms (the t_n's and the u_n's are mutually recursive):

$$t_0 \coloneqq I = \lambda x.x \qquad u_0 \coloneqq t_0 i \qquad r_0 \coloneqq i$$
$$t_{n+1} \coloneqq \lambda z.(y u_n u_n) \text{ with } z \notin \mathtt{fv}(u_n) \cup \{y\} \qquad u_{n+1} \coloneqq t_{n+1} i \qquad r_{n+1} \coloneqq y r_n r_n\,.$$

Proposition 5 *(Exponentially many β_i-steps). For all $n \in \mathbb{N}$, $u_n \to_{\beta_i}^{2^{n+1}-1} r_n$ and r_n is an inert term (and hence β_f-normal).*

Proof. By induction on n. For the base case ($n = 0$), we have $u_0 = (\lambda x.x)i \to_{\beta_i} i = r_0$ where $r_0 = i$ is inert and $2^{0+1} - 1 = 1$. For the inductive case, the *i.h.* says that $u_n \to_{\beta_i}^{2^{n+1}-1} r_n$ and r_n is inert, hence $r_{n+1} = y r_n r_n$ is inert and

$$u_{n+1} = (\lambda z.(y u_n u_n))i \to_{\beta_i} y u_n u_n \to_{\beta_i}^{2^{n+1}-1} y u_n r_n \to_{\beta_i}^{2^{n+1}-1} y r_n r_n = r_{n+1}.$$

Therefore, $u_{n+1} \to_{\beta_i}^{1+2(2^{n+1}-1)} r_{n+1}$ with $1 + 2(2^{n+1} - 1) = 2^{n+2} - 1$. □

Step 2: linearly many β_λ-steps. Define (with $x \neq y$):

$$s_0 \coloneqq I = \lambda x.x \qquad s_{n+1} \coloneqq (\lambda x.\lambda z.(y(xi)(xi)))s_n \text{ with } z \notin \mathtt{fv}(i) \cup \{x, y\}.$$

Proposition 6 *(Linearly many β_λ-steps). For all $n \in \mathbb{N}$, $s_n \to_{\beta_\lambda}^{n} t_n$ and so $s_n i \to_{\beta_\lambda}^{n} t_n i = u_n$.*

Proof. By induction on n. For the base case ($n = 0$), we have $s_0 = t_0$ by definition. In the inductive case ($s_n \to_{\beta_\lambda}^{n} t_n$ by *i.h.*), since all the t_n's are abstractions, we have

$$s_{n+1} \to_{\beta_\lambda}^{n} (\lambda x.\lambda z.(y(xi)(xi)))t_n \to_{\beta_\lambda} \lambda z.(y(t_n i)(t_n i)) = \lambda z.(y u_n u_n) = t_{n+1}.$$ □

Corollary 1 *(Inert length explosion). For all $n \in \mathbb{N}$, $s_n i \to_{\beta_\lambda}^{n} \to_{\beta_i}^{2^{n+1}-1} r_n$.*

Proof. Composing Proposition 5 and Proposition 6. □

Inert length explosion and size of the initial term. The inert length exploding family shows an explosion of β_i-steps (where the argument of each fired β_i-redex is an arbitrary inert term i) with respect to one of the two parameters for complexity analyses, the number of β_λ-steps. In fact, there is an explosion of β_i-steps also with respect to the other parameter, the size of the initial term: an easy induction indeed shows that the size of the initial term $s_n i$ is linear in n.

11. Pitfalls of strong call-by-value

We carefully avoided the definition of Strong CbV, which is a tricky setting with no (reasonable) implementations in the literature. Nonetheless, our results on implementing Open CbV provide some insight into the cost of implementing Strong CbV—overviewed in this section—independently of its exact definition.

Abstractions on-demand: Open CbV is simpler than Strong CbV. We explained in Sect. 4 that Grégoire and Leroy's abstract machine for Coq as described in [27] is unreasonable. Its actual implementation, on the contrary, does not substitute non-variable inert terms, so it is reasonable for Open CbV. None of the versions, however, substitutes abstractions only on-demand (nor, to our knowledge, does any other implementation), despite the fact that it is a necessary optimization in order to have a reasonable implementation of Strong CbV, as we now show. Consider the following size exploding family $\{s_n I\}_{n>0}$ (with $I := \lambda z.z$ and $x \neq y$), from [3]:

$$s_1 := \lambda x.\lambda y.(yxx) \qquad s_{n+1} := \lambda x.(s_n(\lambda y.(yxx))) \qquad r_0 := I \qquad r_{n+1} := \lambda y.(y r_n r_n)$$

Proposition 7 *(Abstraction size explosion). Let $n > 0$. Then $s_n I \to^n_{\beta_\lambda} r_n$. Moreover, $|s_n I| = O(n)$, $|r_n| = \Omega(2^n)$, $s_n I$ and r_n are closed, and r_n is normal.*

Proof. To obtain a simple inductive proof of $s_n I \to^n_{\beta_\lambda} r_n$, we prove by induction on n a more general statement: $s_n r_m \to^n_{\beta_\lambda} r_{n+m}$ for all $n > 0$ and $m \geq 0$ (in particular, for $m = 0$ we have $s_n I \to^n_{\beta_\lambda} r_n$). Note that r_m is an abstraction for all $m \geq 0$.

- *Base case* ($n = 1$): $s_1 r_m = (\lambda x.\lambda y.(yxx))r_m \to_{\beta_\lambda} \lambda y.(y r_m r_m) = r_{m+1}$.
- *Inductive case*: $s_{n+1} r_m = (\lambda x.s_n(\lambda y.(yxx)))r_m \to_{\beta_\lambda} s_n(\lambda y.(y r_m r_m)) = s_n r_{m+1} \to^n_{\beta_\lambda} r_{n+m+1}$, where $s_n r_{m+1} \to^n_{\beta_\lambda} r_{n+m+1}$ by *i.h.*

The proof of the other points of the statement is immediate. □

The family $\{s_n I\}_{n>0}$ is interesting because it is an example of size explosion for Closed, Open and Strong CbV: in all these settings the family inevitably explodes (moreover it is in continuation-passing style and it is typable with simple types). Indeed, any kind of evaluation of $s_n I$ produces 2^n non-applied copies of I (in r_n). In Strong CbV (where CbV evaluation is unrestricted) *all* derivations from $s_n I$ to its normal form r_n have the same length n (and are permutatively equivalent); so, a machine for Strong CbV substituting abstractions whenever (thus not only on-demand) must have an exponential overhead, due to 2^n substitution transitions. If evaluation is weak (*i.e.* it does not go under abstraction, like in Closed and Open CbV) a derivation from $s_n I$ to its normal form r_n still exists—the only possible one, its length is n—and each term in it is closed. Since the substitutions of abstractions (exponential in number) required by the (weak or strong) evaluation of $\{s_n I\}_{n\in\mathbb{N}}$ are all substitutions under non-applied abstractions, closed and open machines never do them anyway: this is the reason why machines for Closed and Open CbV can be reasonable even if they substitute abstractions whenever.

The danger of iterating Open CbV naively. The size exploding example in Proposition 7 also shows that iterating reasonable machines for Open CbV is subtle, as it may induce unreasonable machines for Strong CbV, if done naively. Evaluating Strong CbV by iterating the Easy/BE GLAMOUr (which do *substitute abstractions whenever*, not only on-demand), indeed, induces an *exponential* overhead, while iterating the Fast/BEST GLAMOUr (which does *substitute abstractions only on-demand*) provides an *efficient* implementation: substituting abstractions on-demand is mandatory to get reasonable implementations of Strong CbV by iterating the reasonable implementations of Open CbV we studied here.

12. Conclusions

Let us sum up the lessons learned about abstract machines for Open CbV.

Modular overhead. The overhead of implementing Open CbV is measured with respect to the size $|t_0|$ of the initial term t_0 and the number n of β-steps. We showed that its complexity depends crucially on three choices about substitution.

The first choice is whether to substitute *inert terms that are not variables*. If they are substituted, as in Grégoire and Leroy's machine [27]—here called Open GLAM—then the overhead is exponential in $|t_0|$ because of open size explosion (Proposition 3), and the implementation is then unreasonable.

It turns out, however, that *evaluation* and *substitution of inert terms* commute (Proposition 4), and so it is possible to evaluate without ever substituting inert terms that are not variables. If they are not substituted, as in the machines of the GLAMOUr family, then the overhead becomes polynomial and we get reasonable implementations of Open CbV.

The two other parameters are whether to substitute *variables*, and whether *abstractions* are substituted whenever or only *on-demand*, and they give rise to the following table of machines and reasonable overheads:

	Sub of Abs Whenever	*Sub of Abs On-Demand*
Sub of Variables	Slow GLAMOUr $O((1+n^2)\cdot\|t_0\|^2)$	GLAMOUr $O((1+n^2)\cdot\|t_0\|)$
No Sub of Variables	Easy GLAMOUr $O((1+n)\cdot\|t_0\|^2)$	Fast/Unchaining GLAMOUr $O((1+n)\cdot\|t_0\|)$

The Slow GLAMOUr has been omitted because it is slow and involved, as it requires the labeling mechanism of the (Unchaining) GLAMOUr (see Sect. 9). It is somewhat surprising that the Fast GLAMOUr presented here has the best overhead and it is also the easiest to analyze.

Last, the quadratic dependence on the size $|t_0|$ of the initial term when abstractions are substituted whenever (left column) can be removed by a further optimization, here called *renaming on* β. It does not impact on the right column: the bilinear overhead of the Fast/Unchaining GLAMOUr is optimal.

Minimality of the cost model. Since the substitution of inert terms can be avoided, it is natural to wonder whether one should count at all β-steps whose argument is an inert term. We provided evidence that inert steps should be counted, because they can be exponentially more than the number of non-inert steps. Our evidence is not a proof, it is just a tricky example showing that an implementation reasonable in the number of non-inert steps has to rely on some new implementation technology that, at present, seems to be out of reach.

Pitfalls of Strong CbV. Last, our fine study of how to implement Open CbV in a reasonable way provides insights into how to implement Strong CbV. In particular, we showed that substituting abstraction on-demand is mandatory for reasonable implementations of Strong CbV, even if it is optional for Open CbV. Consequently, seeing Strong CbV as iterated Open CbV is subtle complexity-wise: iterating a reasonable machine for Open CbV may indeed provide only an unreasonable implementation of Strong CbV (if abstractions are substituted whenever).

Acknowledgements

This work has been partially funded by the ANR JCJC grant COCA HOLA (ANR-16-CE40-004-01) and by the EPSRC grant EP/R029121/1 "Typed Lambda-Calculi with Sharing and Unsharing".

The authors are grateful to Claudio Sacerdoti Coen and Andrea Condoluci for valuable discussions, in particular about the renaming on β.

Appendix A. Rewrite theory: definitions, notations, basic results

Given a binary relation $\to_{\mathsf{r}}$ on a set I, the reflexive-transitive (resp. reflexive; transitive; reflexive-transitive and symmetric) closure of $\to_{\mathsf{r}}$ is denoted by $\to_{\mathsf{r}}^*$ (resp. $\to_{\mathsf{r}}^=$; $\to_{\mathsf{r}}^+$; $\simeq_{\mathsf{r}}$). The transpose of $\to_{\mathsf{r}}$ is denoted by ${}_{\mathsf{r}}\!\leftarrow$. A (r-)*derivation* d *from* t *to* u, denoted by $d\colon t \to_{\mathsf{r}}^* u$, is a finite sequence $(t_i)_{0\le i\le n}$ of elements of I (with $n\in\mathbb{N}$) such that $t=t_0$, $u=t_n$ and $t_i \to_{\mathsf{r}} t_{i+1}$ for all $0\le i<n$ (in particular, $t=u$ for $n=0$); we then say that n (also denoted by $|d|_{\mathsf{r}}$ or $|d|$) is the *number of* r*-steps*, or the *length*, of d. We write $t\to_{\mathsf{r}}^n u$ if there is $d\colon t\to_{\mathsf{r}}^* u$ with $|d|=n$. If $\to_{\mathsf{r}}=\to_{\mathsf{r}_1}\cup\to_{\mathsf{r}_2}$ with $\to_{\mathsf{r}_1}\cap\to_{\mathsf{r}_2}=\emptyset$, the *number* $|d|_{\mathsf{r}_k}$ *of* r_k*-steps* (for $k\in\{1,2\}$) in a r-derivation $d=(t_i)_{0\le i\le n}$ is the number of t_i's in d such that $t_i\to_{\mathsf{r}_k} t_{i+1}$. We say that:

- $t\in I$ is r*-normal* or a r*-normal form* if $t\not\to_{\mathsf{r}} u$ for all $u\in I$; $u\in I$ is a r*-normal form of* t if u is r-normal and $t\to_{\mathsf{r}}^* u$;
- $t\in I$ is r*-normalizable* if $t\to_{\mathsf{r}}^* u$ for some r-normal $u\in I$; t r*-diverges* if there is an infinite sequence $(t_i)_{i\in\mathbb{N}}$ such that $t_0=t$ and $t_i\to_{\mathsf{r}} t_{i+1}$ for all $i\in\mathbb{N}$, otherwise t is *strongly* r*-normalizable*;
- a r-derivation $d\colon t\to_{\mathsf{r}}^* u$ is (r-)*normalizing* if u is r-normal;
- $\to_{\mathsf{r}}$ is *strongly normalizing* (or *terminates*) if every $t\in I$ is strongly r-normalizable;
- $\to_{\mathsf{r}}$ is *deterministic* if, for all $t,u,s\in I$, $s\,{}_{\mathsf{r}}\!\leftarrow t\to_{\mathsf{r}} u$ implies $u=s$; $\to_{\mathsf{r}}$ is *quasi-diamond* if, for all $t,u,s\in I$ such that $s\,{}_{\mathsf{r}}\!\leftarrow t\to_{\mathsf{r}} u$ and $u\neq s$, there is $r\in I$ such that $s\to_{\mathsf{r}} r\,{}_{\mathsf{r}}\!\leftarrow u$; $\to_{\mathsf{r}}$ is *confluent* if $\to_{\mathsf{r}}^*$ is quasi-diamond.

Let $\to_{\mathsf{r}_1}$ and $\to_{\mathsf{r}_2}$ be binary relations on a set I. Composition of relations is denoted by juxtaposition: for instance, $t\to_{\mathsf{r}_1}\to_{\mathsf{r}_2} u$ means that there is $s\in I$ such that $t\to_{\mathsf{r}_1} s\to_{\mathsf{r}_2} u$. We say that $\to_{\mathsf{r}_1}$ and $\to_{\mathsf{r}_2}$ *strongly commute* if, for any $t,u,s\in I$ such that $u\,{}_{\mathsf{r}_1}\!\leftarrow t\to_{\mathsf{r}_2} s$, one has $u\neq s$ and there is $r\in I$ such that $u\to_{\mathsf{r}_2} r\,{}_{\mathsf{r}_1}\!\leftarrow s$. Note that if $\to_{\mathsf{r}_1}$ and $\to_{\mathsf{r}_2}$ strongly commute and $\to=\to_{\mathsf{r}_1}\cup\to_{\mathsf{r}_2}$, then for any derivation $d\colon t\to^* u$ the sizes $|d|_{\mathsf{r}_1}$ and $|d|_{\mathsf{r}_2}$ are uniquely determined.

The next proposition collects some basic and well-known results of rewriting theory, we use them implicitly in the paper.

Proposition 8 *([35]). Let* $\to_{\mathsf{r}}$ *be a binary relation on a set* I.

1. *If* $\to_{\mathsf{r}}$ *is confluent then:*
   (a) *every* r*-normalizable term has a unique* r*-normal form;*
   (b) *for all* $t,u\in I$, $t\simeq_{\mathsf{r}} u$ *if and only if there is* $s\in I$ *such that* $t\to_{\mathsf{r}}^* s\,{}_{\mathsf{r}}^*\!\leftarrow u$.

2. *If $\to_{\mathsf{r}}$ is quasi-diamond then $\to_{\mathsf{r}}$ is confluent and, for any $t \in I$:*

   (a) *all normalizing* r*-derivations from t have the same length;*
   (b) *t is strongly* r*-normalizable if and only if t is* r*-normalizable.*

The rewriting theory of the fireball calculus λ_{fire} is very well behaved: for instance, the reductions $\to_{\beta_i}$, $\to_{\beta_\lambda}$ and $\to_{\beta_f}$ (which are weak, *i.e.* do not reduce under abstractions) are quasi-diamond. Other good operational properties of λ_{fire} are summarized in Proposition 2 on p. 5. Its proof rests on the following more informative version of the Hindley–Rosen Lemma [16, Proposition 3.3.5.(i)]:

Lemma 15 *(Strong Hindley–Rosen). Let $\to_{\mathsf{r}} = \to_{\mathsf{r}_1} \cup \to_{\mathsf{r}_2}$ be a binary relation on a set I, where $\to_{\mathsf{r}_1}$ and $\to_{\mathsf{r}_2}$ are quasi-diamond and strongly commute. Then, $\to_{\mathsf{r}}$ is quasi-diamond and, for any $t \in I$ and any normalizing* r*-derivations d and e from t, one has $|d|_{\mathsf{r}} = |e|_{\mathsf{r}}$, $|d|_{\mathsf{r}_1} = |e|_{\mathsf{r}_1}$ and $|d|_{\mathsf{r}_2} = |e|_{\mathsf{r}_2}$.*

The proof of Lemma 15 is just a more accurate reading of the proof in [16, Proposition 3.3.5.(i)].

Appendix B. Omitted proofs

For the sake of completeness, we collect here some proofs of minor lemmas stated in the main text. These proofs are not included in the body of the paper because they are straightforward or trivially obtained by very similar ones already present in the main text.

B.1. Proofs of Section 2 (The fireball calculus)

Circumventing open size explosion. To prove that the substitution of inert terms can be avoided (commutation of evaluation and substitution of inert terms, Proposition 4) we need two auxiliary lemmas about substitution, fireballs, and reductions.

Lemma 2 *(Fireballs are closed under substitution and anti-substitution of inert terms).*[11] *Let t be a term and i be an inert term.*

1. *$t\{x\leftarrow i\}$ is an abstraction if and only if t is an abstraction;*
2. *$t\{x\leftarrow i\}$ is an inert term if and only if t is an inert term;*
3. *$t\{x\leftarrow i\}$ is a fireball if and only if t is a fireball.*

Proof. 1. If $t\{x\leftarrow i\} = \lambda y.s$ then t is neither a variable nor an application, otherwise $t\{x\leftarrow i\}$ would be, respectively, an inert term or an application, and not an abstraction. Therefore, t is an abstraction.

Conversely, if $t = \lambda y.s$ then we can suppose without loss of generality that $y \notin \mathtt{fv}(i) \cup \{x\}$ and so $t\{x\leftarrow i\} = \lambda y.(s\{x\leftarrow i\})$, which is an abstraction.

2. ($\Rightarrow$): By induction on the inert structure of $t\{x\leftarrow i\}$. Cases:

   - *Variable*, *i.e.* $t\{x\leftarrow i\} = y$ (possibly $x = y$). Then t is neither an application nor an abstraction, otherwise $t\{x\leftarrow i\}$ would be, respectively, an application or an abstraction. Therefore, t is a variable and hence an inert term.
   - *Compound Inert*, *i.e.* $t\{x\leftarrow i\} = i'f$. Then t is not an abstraction, otherwise $t\{x\leftarrow i\}$ would be an abstraction and not an inert term. If t is a variable then it is inert. Otherwise it is an application $t = us$, and hence $u\{x\leftarrow i\} = i'$ and $s\{x\leftarrow i\} = f$; by *i.h.*, u is an inert term; according to the definition of fireball, there are two subcases for f:

     (a) f is an abstraction; then by Point 1 s is an abstraction;
     (b) f is an inert term; then by *i.h.* s is an inert term.

     In both subcases s is a fireball, and so $t = us$ is an inert term.

   ($\Leftarrow$): By induction on the inert structure of t. Cases:

   - *Variable*, *i.e.* either $t = x$ or $t = y \neq x$: in the first subcase $t\{x\leftarrow i\} = i$, in the second subcase $t\{x\leftarrow i\} = y$; in both subcases $t\{x\leftarrow i\}$ is an inert term.
   - *Compound Inert*, *i.e.* $t = i'f$. Then $t\{x\leftarrow i\} = i'\{x\leftarrow i\}f\{x\leftarrow i\}$. By *i.h.*, $i'\{x\leftarrow i\}$ is an inert term. Concerning f, there are two subcases, according to the definition of fireball:

     (a) f is an abstraction; then by Point 1 $f\{x\leftarrow i\}$ is an abstraction;

[11] Stated at p. 7.

(b) f is an inert term; then by *i.h.* $f\{x\leftarrow i\}$ is an inert term.

In both subcases $f\{x\leftarrow i\}$ is a fireball, and hence $t\{x\leftarrow i\} = i'\{x\leftarrow i\}f\{x\leftarrow i\}$ is an inert term.

3. Immediate consequence of Lemmas 2.1-2, since every fireball is either an abstraction or an inert term. □

Lemma 3 *(Substitution of inert terms does not create β_f-redexes).*[12] *Let t, u be terms and i be an inert term. There is a term s such that:*

1. *if $t\{x\leftarrow i\} \to_{\beta_\lambda} u$ then $t \to_{\beta_\lambda} s$ and $s\{x\leftarrow i\} = u$;*
2. *if $t\{x\leftarrow i\} \to_{\beta_i} u$ then $t \to_{\beta_i} s$ and $s\{x\leftarrow i\} = u$.*

Proof. Since $\to_{\beta_f} = \to_{\beta_\lambda} \cup \to_{\beta_i}$, we prove simultaneously both points, by induction on the definition of $t\{x\leftarrow i\} \to_{\beta_f} u$ (*i.e.* by induction on the evaluation context closing the fired β_f-redex). Cases:

- *Step at the root, i.e.* $t\{x\leftarrow i\} := (\lambda y.r')q' \mapsto_{\beta_f} r'\{y\leftarrow q'\} =: u$. Since t is not a fireball (otherwise, by Lemma 2.3, $t\{x\leftarrow i\}$ would be a fireball and hence β_f-normal, according to Proposition 1.1), it has the form $t = pq$ with $p\{x\leftarrow i\} = \lambda y.r'$ and $q\{x\leftarrow i\} = q'$. By Lemma 2.1, $p = \lambda y.r$ with $r' = r\{x\leftarrow i\}$ (we can suppose without loss of generality that $y \notin \mathtt{fv}(i) \cup \{x\}$). There are two subcases:

  1. *Abstraction step, i.e.* $t\{x\leftarrow i\} = (\lambda y.r')q' \mapsto_{\beta_\lambda} r'\{y\leftarrow q'\} = u$. By Lemma 2.1, q is an abstraction, as $q' = q\{x\leftarrow i\}$ is an abstraction. Then, $t = (\lambda y.r)q \mapsto_{\beta_\lambda} r\{y\leftarrow q\}$ and $s := r\{y\leftarrow q\}$ verifies the claim, since $s\{x\leftarrow i\} = (r\{y\leftarrow q\})\{x\leftarrow i\} = r\{x\leftarrow i\}\{y\leftarrow q\{x\leftarrow i\}\} = u$.
  2. *Inert step, i.e.* $t\{x\leftarrow i\} = (\lambda y.r')q' \mapsto_{\beta_i} r'\{y\leftarrow q'\} = u$. Analogous to the abstraction subcase, just replace *abstraction* with *inert term*, the use of Lemma 2.1 with the use of Lemma 2.2, and $\mapsto_{\beta_\lambda}$ with $\mapsto_{\beta_i}$.

- *Application left, i.e.* $t\{x\leftarrow i\} := r'q' \to_{\beta_f} p'q' =: u$ with $r' \to_{\beta_f} p'$. Since t is not a fireball (otherwise, by Lemma 2.3, $t\{x\leftarrow i\}$ would be a fireball and hence β_f-normal, according to Proposition 1.1), it has the form $t = rq$ with $r\{x\leftarrow i\} = r'$ and $q\{x\leftarrow i\} = q'$. There are two subcases (for $r' \to_{\beta_f} p'$):

  1. *Abstraction step, i.e.* $r\{x\leftarrow i\} \to_{\beta_\lambda} p'$. By *i.h.* there is a term p such that $p' = p\{x\leftarrow i\}$ and $r \to_{\beta_\lambda} p$. Then $s := pq$ satisfies the statement, as $t = rq \to_{\beta_\lambda} pq = s$ and $s\{x\leftarrow i\} = (pq)\{x\leftarrow i\} = p\{x\leftarrow i\}q\{x\leftarrow i\} = u$.
  2. *Inert step, i.e.* $r\{x\leftarrow i\} \to_{\beta_i} p'$. Analogous to the abstraction subcase, just replace $\to_{\beta_\lambda}$ with $\to_{\beta_i}$.

- *Application right, i.e.* $t\{x\leftarrow i\} := q'r' \to_{\beta_f} q'p' =: u$ with $r' \to_{\beta_f} p'$. Analogous to the previous case, just swap left and right. □

B.2. Proofs of Section 5 (Easy GLAMOUr)

In order to prove that the Easy GLAMOUr implements the right-to-left strategy in the fireball calculus (Theorem 2), we use the following qualitative invariants of the Easy GLAMOUr.

Lemma 5 *(Easy GLAMOUr invariants).*[13] *Let $s = (D, \overline{t}, \pi, E)$ be a reachable state of an Easy GLAMOUr execution. Then:*

1. Name:
   - *(a)* Explicit substitution*: if $E = E'[x\leftarrow\phi]E''$ then the variable x is fresh with respect to ϕ and E'';*
   - *(b)* Abstraction*: if $\lambda x.\overline{u}$ is a subterm of D, $\overline{t}$, π or E, then the variable x may occur only in $\overline{u}$.*
2. Fireball item*: for every item ϕ in π, in E or in any stack in D, one has that $\underline{\phi}$ and $\underline{\phi}\downarrow_E$ are inert terms if $\phi = x@\pi'$, and abstractions otherwise.*
3. Contextual decoding*: $R_s = \underline{D}\langle\underline{\pi}\rangle\downarrow_E$ is a right context.*

Proof. By induction on the length of the execution leading to the reachable state s. In an initial state all the invariants trivially hold. For a non-empty execution, the proof for each invariant is by case analysis on the last transition, using the *i.h.*

1. *Name.* Cases:
   i. $s' = (D, \overline{t}\overline{u}, \pi, E) \rightsquigarrow_{c1} (D : \overline{t}\Diamond\pi, \overline{u}, \epsilon, E) = s$. Both points follow immediately from the *i.h.*

[12] Stated at p. 7.
[13] Stated at p. 14.

ii. $s' = (D : \overline{t}\Diamond\pi, \lambda x.\overline{u}, \epsilon, E) \rightsquigarrow_{c_2} (D, \overline{t}, (\lambda x.\overline{u}@\epsilon) : \pi, E) = s$. Both points follow immediately from the *i.h.*
iii. $s' = (D : \overline{t}\Diamond\pi, x, \pi', E) \rightsquigarrow_{c_3} (D, \overline{t}, (x@\pi') : \pi, E) = s$ with $E(x) = \bot$ or $E(x) = y@\pi''$. Both points follow from the *i.h.*
iv. $s' = (D, \lambda x.\overline{t}, \phi : \pi, E) \rightsquigarrow_{\beta} (D, \overline{t}, \pi, [x\leftarrow\phi]E) = s$. Point 1a for the new entry $[x\leftarrow\phi]$ in the environment of s follows from the *i.h.* for Point 1b, for the other entries from the *i.h.* for Point 1a. Point 1b follows from its *i.h.*
v. $s' = (D, x, \pi, E) \rightsquigarrow_{s} (D, (\lambda y.\overline{u})^{\alpha}, \pi, E) = s$ where $E = E_1[x\leftarrow\lambda y.\overline{u}@\epsilon]E_2$. Point 1a follows from its *i.h.*; Point 1b for the new code of s is guaranteed by the α-renaming operation $(\lambda y.\overline{u})^{\alpha}$, the rest follows from its *i.h.*

2. *Fireball item.* Cases:

(a) $s' = (D, \overline{t}\overline{u}, \pi, E) \rightsquigarrow_{c_1} (D : \overline{t}\Diamond\pi, \overline{u}, \epsilon, E) = s$. The invariant follows directly from the *i.h.* (as the stack π of the entry $\overline{t}\Diamond\pi$ in the dump of s is the stack of s').
(b) $s' = (D : \overline{t}\Diamond\pi, \lambda x.\overline{u}, \epsilon, E) \rightsquigarrow_{c_2} (D, \overline{t}, (\lambda x.\overline{u}@\epsilon) : \pi, E) = s$. For the item $\lambda x.\overline{u}@\epsilon$, we have that $\underline{\lambda x.\overline{u}@\epsilon} = \lambda x.\underline{\overline{u}}$ and $\underline{\lambda x.\overline{u}@\epsilon}\downarrow_E = (\lambda x.\underline{\overline{u}})\downarrow_E = \lambda x.(\underline{\overline{u}}\downarrow_E)$ are abstractions (the last equality holds by Lemma 5.1b). For all other items in s, the invariant follows from the *i.h.* (as the tail π of the stack of s is the stack of the entry $\overline{t}\Diamond\pi$ in the dump of s').
(c) $s' = (D : \overline{t}\Diamond\pi, x, \pi', E) \rightsquigarrow_{c_3} (D, \overline{t}, (x@\pi') : \pi, E) = s$ with $E(x) = \bot$ or $E(x) = y@\pi''$. For the item $x@\pi'$, we have that $\underline{x@\pi'} = \langle x\rangle\underline{\pi'}$ is inert because, by *i.h.*, $\underline{\psi}$ is a fireball for every item ψ in π'. As for $\underline{x@\pi'}\downarrow_E$, there are two subcases:

i. $E(x) = y@\pi''$ *i.e.* $E := E_1[x\leftarrow y@\pi'']E_2$. By Lemma 5.1a, every entry in E binds a different variable, and x and the variables bound in E_1 are fresh with respect to $y@\pi''$, so $x\downarrow_E = x\downarrow_{E_1[x\leftarrow y@\pi'']E_2} = x\{x\leftarrow\underline{y@\pi''}\}\downarrow_{E_2} = \underline{y@\pi''}\downarrow_{E_2} = \underline{y@\pi''}\downarrow_E$, which by *i.h.* is inert. The *i.h.* also says that $\underline{\psi}\downarrow_E$ is a fireball for every item ψ in π'. Thus, $\underline{x@\pi'}\downarrow_E = \langle x\downarrow_E\rangle(\underline{\pi'}\downarrow_E) = \langle\underline{y@\pi''}\downarrow_E\rangle(\underline{\pi'}\downarrow_E)$ is inert.
ii. $E(x) = \bot$. Similar to the previous case. By hypothesis, we have $x\downarrow_E = x$. As before, by *i.h.* $\underline{\psi}\downarrow_E$ is a fireball for every item ψ in π'. So, $\underline{x@\pi'}\downarrow_E = \langle x\downarrow_E\rangle(\underline{\pi'}\downarrow_E) = \langle x\rangle(\underline{\pi'}\downarrow_E)$ is an inert term.

For all other items in s the invariant follows from the *i.h.* (as the tail π of the stack of s is the stack of the entry $\overline{t}\Diamond\pi$ in the dump of s').
(d) $s' = (D, \lambda x.\overline{t}, \phi : \pi, E) \rightsquigarrow_{\beta} (D, \overline{t}, \pi, [x\leftarrow\phi]E) = s$. By Lemma 5.1b, x may occur only in $\overline{t}$. Thus the substitution $\downarrow_{[x\leftarrow\phi]E}$ acts exactly as $\downarrow_E$ on every item in s; therefore, the invariant follows from the *i.h.* for every item in s (as the item ϕ in the new entry in the environment of s is in the stack of s').
(e) $s' = (D, x, \pi, E) \rightsquigarrow_{s} (D, (\lambda y.\overline{u})^{\alpha}, \pi, E) = s$ where $E = E_1[x\leftarrow\lambda y.\overline{u}@\epsilon]E_2$. The invariant follows directly from the *i.h.* (as the code $(\lambda y.\overline{u})^{\alpha}$ of s is the only component that changes from s').

3. *Contextual decoding.* Cases: (we use the fact that, given two contexts C and C', the composition $C\langle C'\rangle$—obtained from C by replacing its hole with C'—is a right context iff C and C' are so; the proofs of both directions are easy inductions)

(a) $s' = (D, \overline{t}\overline{u}, \pi, E) \rightsquigarrow_{c_1} (D : \overline{t}\Diamond\pi, \overline{u}, \epsilon, E) = s$. By *i.h.*, $R_{s'} = \underline{D}\langle\underline{\pi}\rangle\downarrow_E$ is a right context, as well as $\underline{\overline{u}}\downarrow_E\langle\cdot\rangle$. Then their composition $(\underline{D}\langle\underline{\pi}\rangle\downarrow_E)\langle\underline{\overline{u}}\downarrow_E\langle\cdot\rangle\rangle = \underline{D}\langle\langle\underline{\overline{u}}\langle\cdot\rangle\rangle\underline{\pi}\rangle\downarrow_E = R_s$ is a right context.
(b) $s' = (D : \overline{t}\Diamond\pi, \lambda x.\overline{u}, \epsilon, E) \rightsquigarrow_{c_2} (D, \overline{t}, (\lambda x.\overline{u}@\epsilon) : \pi, E) = s$. By *i.h.*, $R_{s'} = \underline{D : \overline{t}\Diamond\pi}\downarrow_E = \underline{D}\langle\langle\underline{\overline{t}}\langle\cdot\rangle\rangle\underline{\pi}\rangle\downarrow_E = (\underline{D}\langle\underline{\pi}\rangle\downarrow_E)\langle\underline{\overline{t}}\downarrow_E\langle\cdot\rangle\rangle$ is a right context, which implies that $\underline{D}\langle\underline{\pi}\rangle\downarrow_E$ is one such context. So, $R_s = \underline{D}\langle\underline{(\lambda x.\overline{u}@\epsilon) : \pi}\rangle\downarrow_E = \underline{D}\langle\langle\langle\cdot\rangle\lambda x.\underline{\overline{u}}\rangle\underline{\pi}\rangle\downarrow_E = (\underline{D}\langle\underline{\pi}\rangle\downarrow_E)\langle\langle\cdot\rangle\lambda x.(\underline{\overline{u}}\downarrow_E)\rangle$ is a right context because it is the composition of right contexts, in that $\lambda x.(\underline{\overline{u}}\downarrow_E)$ is a fireball.
(c) $s' = (D : \overline{t}\Diamond\pi, x, \pi', E) \rightsquigarrow_{c_3} (D, \overline{t}, (x@\pi') : \pi, E) = s$ with $E(x) = \bot$ or $E(x) = y@\pi''$. By *i.h.*, $R_{s'} = \underline{D : \overline{t}\Diamond\pi}\langle\underline{\pi'}\rangle\downarrow_E = \underline{D}\langle\langle\underline{\overline{t}}\langle\underline{\pi'}\rangle\rangle\underline{\pi}\rangle\downarrow_E = (\underline{D}\langle\underline{\pi}\rangle\downarrow_E)\langle\underline{\overline{t}}\langle\underline{\pi'}\rangle\downarrow_E\rangle$ is a right context, which implies that $\underline{D}\langle\underline{\pi}\rangle\downarrow_E$ is one such context as well. Therefore, $R_s = \underline{D}\langle\underline{(x@\pi') : \pi}\rangle\downarrow_E = \underline{D}\langle\langle\cdot\rangle\underline{x@\pi'}\rangle\underline{\pi}\rangle\downarrow_E = (\underline{D}\langle\underline{\pi}\rangle\downarrow_E)\langle\langle\cdot\rangle\underline{x@\pi'}\downarrow_E\rangle$ is a right context because it is the composition of right contexts, given that $\underline{x@\pi'}\downarrow_E$ is a fireball by Lemma 5.2.
(d) $s' = (D, \lambda x.\overline{t}, \phi : \pi, E) \rightsquigarrow_{\beta} (D, \overline{t}, \pi, [x\leftarrow\phi]E) = s$. By *i.h.*, $R_{s'} = \underline{D}\langle\underline{\phi : \pi}\rangle\downarrow_E = \underline{D}\langle\langle\langle\cdot\rangle\underline{\phi}\rangle\underline{\pi}\rangle\downarrow_E = (\underline{D}\langle\underline{\pi}\rangle\downarrow_E)\langle\langle\cdot\rangle\underline{\phi}\downarrow_E\rangle$ is a right context, which implies that $\underline{D}\langle\underline{\pi}\rangle\downarrow_E$ is one such context as well. Now, $R_s = \underline{D}\langle\underline{\pi}\rangle\downarrow_{[x\leftarrow\phi]E} = \underline{D}\langle\underline{\pi}\rangle\downarrow_E$ because by Lemma 5.1b x may occur only in $\overline{t}$, and so the substitution $\downarrow_{[x\leftarrow\phi]E}$ acts on every code in D and π exactly as $\downarrow_E$.
(e) $s' = (D, x, \pi, E[x\leftarrow\lambda y.\overline{u}@\epsilon]E') \rightsquigarrow_{s} (D, (\lambda y.\overline{u})^{\alpha}, \pi, E[x\leftarrow\lambda y.\overline{u}@\epsilon]E') = s$. The invariant follows from the *i.h.* because $R_{s'} = R_s$, as the only component that changes is the current code. □

B.3. Proofs of Section 7 (Fast GLAMOUr)

In order to prove that the Fast GLAMOUr implements the right-to-left strategy in the fireball calculus (Theorem 4), we proceed like for the Easy GLAMOUr: first we prove the invariants, and then we use them to prove that the Fast GLAMOUr forms an implementation system (Definition 2) with respect to right-to-left evaluation $\rightarrow_{x\beta_f}$ in the fireball calculus (via the decoding). The differences are minimal, but we include detailed proofs for the sake of completeness.

Lemma 11 *(Fast GLAMOUr invariants).*[14] *Let $s = (D, \overline{t}, \pi, E)$ be a reachable state of a Fast GLAMOUr execution. Then:*

1. Name:
   (a) Explicit substitution*: if $E = E'[x\leftarrow\phi]E''$ then the variable x is fresh with respect to ϕ and E'';*
   (b) Abstraction*: if $\lambda x.\overline{u}$ is a subterm of D, $\overline{t}$, π or E, then the variable x may occur only in $\overline{u}$.*
2. Fireball item*: for every item ϕ in π, in E or in any stack in D, one has that $\underline{\phi}$ and $\underline{\phi}\downarrow_E$ are: inert terms if $\phi = x@\pi'$ and either $E(x) = \bot$ or $E(x) = y@\pi''$; abstractions otherwise.*
3. Contextual decoding*: $R_s = \underline{D}\langle\underline{\pi}\rangle\downarrow_E$ is a right context.*

Proof. By induction on the length of the execution leading to the reachable state s. In an initial state all the invariants trivially hold. For a non-empty execution, the proof for each invariant is by case analysis on the last transition, using the *i.h.*

1. *Name.* Cases:
   i. $s' = (D, \overline{t}\overline{u}, \pi, E) \rightsquigarrow_{c_1} (D : \overline{t}\Diamond\pi, \overline{u}, \epsilon, E) = s$. Both points follow immediately from the *i.h.*
   ii. $s' = (D : \overline{t}\Diamond\pi, \lambda x.\overline{u}, \epsilon, E) \rightsquigarrow_{c_2} (D, \overline{t}, (\lambda x.\overline{u}@\epsilon) : \pi, E) = s$. Both points follow immediately from the *i.h.*
   iii. $s' = (D : \overline{t}\Diamond\pi, x, \pi', E) \rightsquigarrow_{c_3} (D, \overline{t}, (x@\pi') : \pi, E) = s$ with $E(x) = \bot$ or $E(x) = y@\pi''$ or $(E(x) = \lambda y.\overline{u}@\epsilon$ and $\pi' = \epsilon)$. Both points follow immediately from the *i.h.*
   iv. $s' = (D, \lambda x.\overline{t}, (y@\epsilon) : \pi, E) \rightsquigarrow_{\beta_1} (D, \overline{t}\{x\leftarrow y\}, \pi, E) = s$. Both points follow immediately from the *i.h.*
   v. $s' = (D, \lambda x.\overline{t}, \phi : \pi, E) \rightsquigarrow_{\beta_2} (D, \overline{t}, \pi, [x\leftarrow\phi]E) = s$ with $\phi \neq y@\epsilon$. Point 1a for the new entry $[x\leftarrow\phi]$ in the environment of s follows from the *i.h.* for Point 1b, for the other entries from the *i.h.* for Point 1a. Point 1b follows from its *i.h.*
   vi. $s' = (D, x, \phi{:}\pi, E) \rightsquigarrow_{s} (D, (\lambda y.\overline{u})^{\alpha}, \phi{:}\pi, E) = s$ where $E = E_1[x\leftarrow\lambda y.\overline{u}@\epsilon]E_2$. Point 1a follows from its *i.h.*; Point 1b for the new code of s is guaranteed by the α-renaming operation $(\lambda y.\overline{u})^{\alpha}$, the rest follows from its *i.h.*
2. *Fireball item.* Cases:
   (a) $s' = (D, \overline{t}\overline{u}, \pi, E) \rightsquigarrow_{c_1} (D : \overline{t}\Diamond\pi, \overline{u}, \epsilon, E) = s$. The invariant follows directly from the *i.h.* (as the stack π of the entry $\overline{t}\Diamond\pi$ in the dump of s is the stack of s').
   (b) $s' = (D : \overline{t}\Diamond\pi, \lambda x.\overline{u}, \epsilon, E) \rightsquigarrow_{c_2} (D, \overline{t}, (\lambda x.\overline{u}@\epsilon) : \pi, E) = s$. For the item $\lambda x.\overline{u}@\epsilon$, we have that $\underline{\lambda x.\overline{u}@\epsilon} = \lambda x.\underline{\overline{u}}$ and $\underline{\lambda x.\overline{u}@\epsilon}\downarrow_E = (\lambda x.\underline{\overline{u}})\downarrow_E = \lambda x.(\underline{\overline{u}}\downarrow_E)$ are abstractions (the last equality holds by Lemma 11.1b). For all other items in s, the invariant follows from the *i.h.* (as the tail π of the stack of s is the stack of the entry $\overline{t}\Diamond\pi$ in the dump of s').
   (c) $s' = (D : \overline{t}\Diamond\pi, x, \pi', E) \rightsquigarrow_{c_3} (D, \overline{t}, (x@\pi') : \pi, E) = s$ with $E(x) = \bot$ or $E(x) = y@\pi''$ or $(E(x) = \lambda y.\overline{u}@\epsilon$ and $\pi' = \epsilon)$. For the item $x@\pi'$, we have that $\underline{x@\pi'} = \langle x\rangle\underline{\pi'}$ is an inert term because, by *i.h.*, $\underline{\psi}$ is a fireball for every item ψ in π'. Concerning $\underline{x@\pi'}\downarrow_E$, there are three subcases:
      i. $E(x) = y@\pi''$ *i.e.* $E := E_1[x\leftarrow y@\pi'']E_2$. By Lemma 11.1a, every entry in E binds a different variable, so $x\downarrow_E = x\downarrow_{E_1[x\leftarrow y@\pi'']E_2} = x\{x\leftarrow\underline{y@\pi''}\}\downarrow_{E_2} = \underline{y@\pi''}\downarrow_{E_2} = \underline{y@\pi''}\downarrow_E$, which by *i.h.* is inert. The *i.h.* also says that $\underline{\psi}\downarrow_E$ is a fireball for every item ψ in π'. Thus, $\underline{x@\pi'}\downarrow_E = \langle x\downarrow_E\rangle(\underline{\pi'}\downarrow_E) = \langle\underline{y@\pi''}\downarrow_E\rangle(\underline{\pi'}\downarrow_E)$ is inert.
      ii. $E(x) = \bot$. Similar to the previous case. By hypothesis, we have $x\downarrow_E = x$. As before, by *i.h.* $\underline{\psi}\downarrow_E$ is a fireball for every item ψ in π'. So, $\underline{x@\pi'}\downarrow_E = \langle x\downarrow_E\rangle(\underline{\pi'}\downarrow_E) = \langle x\rangle(\underline{\pi'}\downarrow_E)$ is an inert term.
      iii. $E(x) = \lambda y.\overline{u}@\epsilon$ *(i.e.* $E = E_1[x\leftarrow\lambda y.\overline{u}@\epsilon]E_2$*)* and $\pi' = \epsilon$. Then $\underline{x@\pi'} = x$. By Lemma 11.1a, every entry in E binds a different variable, and x and the variables bound in E_1 are fresh with respect to $\lambda y.\overline{u}@\epsilon$, so $\underline{x@\pi'}\downarrow_E = x\downarrow_{E_1[x\leftarrow\lambda y.\overline{u}@\epsilon]E_2} = x\{x\leftarrow\underline{\lambda y.\overline{u}@\epsilon}\}\downarrow_{E_2} = \underline{\lambda y.\overline{u}@\epsilon}\downarrow_{E_2} = (\lambda y.\underline{\overline{u}})\downarrow_E =$ (by Lemma 11.1b) $\lambda y.(\underline{\overline{u}}\downarrow_E)$ is an abstraction.

      For all other items in s the invariant follows from the *i.h.* (as the tail π of the stack of s is the stack of the entry $\overline{t}\Diamond\pi$ in the dump of s').
   (d) $s' = (D, \lambda x.\overline{t}, (y@\epsilon) : \pi, E) \rightsquigarrow_{\beta_1} (D, \overline{t}\{x\leftarrow y\}, \pi, E) = s$. For all items in s, the invariant follows directly from the *i.h.* (as they are items in s' as well).
   (e) $s' = (D, \lambda x.\overline{t}, \phi : \pi, E) \rightsquigarrow_{\beta_2} (D, \overline{t}, \pi, [x\leftarrow\phi]E) = s$ with $\phi \neq y@\epsilon$. By Lemma 11.1b, x may occur only in $\overline{t}$. Thus the substitution $\downarrow_{[x\leftarrow\phi]E}$ acts exactly as $\downarrow_E$ on every item in s; therefore, the invariant follows from the *i.h.* for every item in s (as the item ϕ in the new entry in the environment of s is in the stack of s').
   (f) $s' = (D, x, \phi{:}\pi, E) \rightsquigarrow_{s} (D, (\lambda y.\overline{u})^{\alpha}, \phi{:}\pi, E) = s$ where $E = E_1[x\leftarrow\lambda y.\overline{u}@\epsilon]E_2$. The invariant follows directly from the *i.h.* (as the code $(\lambda y.\overline{u})^{\alpha}$ of s is the only component that changes from s').

[14] Stated at p. 19.

3. *Contextual decoding.* Cases:

   (a) $s' = (D, \overline{t}\overline{u}, \pi, E) \rightsquigarrow_{c_1} (D : \overline{t}\Diamond\pi, \overline{u}, \epsilon, E) = s$. By *i.h.*, $R_{s'} = \underline{D}\langle\underline{\pi}\rangle\downarrow_E$ is a right context, as well as $\overline{u}\downarrow_E\langle\cdot\rangle$. Then their composition $(\underline{D}\langle\underline{\pi}\rangle\downarrow_E)\langle\overline{u}\downarrow_E\langle\cdot\rangle\rangle = \underline{D}\langle\langle\overline{u}\langle\cdot\rangle\rangle\underline{\pi}\rangle\downarrow_E = R_s$ is a right context.

   (b) $s' = (D : \overline{t}\Diamond\pi, \lambda x.\overline{u}, \epsilon, E) \rightsquigarrow_{c_2} (D, \overline{t}, (\lambda x.\overline{u}@\epsilon) : \pi, E) = s$. By *i.h.*, $R_{s'} = \underline{D : \overline{t}\Diamond\pi}\downarrow_E = \underline{D}\langle\langle\overline{t}\langle\cdot\rangle\rangle\underline{\pi}\rangle\downarrow_E = (\underline{D}\langle\underline{\pi}\rangle\downarrow_E)\langle\overline{t}\downarrow_E\langle\cdot\rangle\rangle$ is a right context, which implies that $\underline{D}\langle\underline{\pi}\rangle\downarrow_E$ is one such context. So, $R_s = \underline{D}\langle\underline{(\lambda x.\overline{u}@\epsilon) : \pi}\rangle\downarrow_E = \underline{D}\langle\langle\langle\cdot\rangle\lambda x.\overline{u}\rangle\underline{\pi}\rangle\downarrow_E = (\underline{D}\langle\underline{\pi}\rangle\downarrow_E)\langle\langle\cdot\rangle\lambda x.(\overline{u}\downarrow_E)\rangle$ is a right context because it is the composition of right contexts, in that $\lambda x.(\overline{u}\downarrow_E)$ is a fireball.

   (c) $s' = (D : \overline{t}\Diamond\pi, x, \pi', E) \rightsquigarrow_{c_3} (D, \overline{t}, (x@\pi') : \pi, E) = s$ with $E(x) = \bot$ or $E(x) = y@\pi''$ or $(E(x) = \lambda y.\overline{u}@\epsilon$ and $\pi' = \epsilon)$. By *i.h.*, $R_{s'} = \underline{D : \overline{t}\Diamond\pi\langle\underline{\pi'}\rangle}\downarrow_E = \underline{D}\langle\langle\overline{t}\langle\underline{\pi'}\rangle\rangle\underline{\pi}\rangle\downarrow_E = (\underline{D}\langle\underline{\pi}\rangle\downarrow_E)\langle\overline{t}\langle\underline{\pi'}\rangle\downarrow_E\rangle$ is a right context, which implies that $\underline{D}\langle\underline{\pi}\rangle\downarrow_E$ is one such context as well. Then $R_s = \underline{D}\langle\underline{(x@\pi') : \pi}\rangle\downarrow_E = \underline{D}\langle\langle\langle\cdot\rangle\underline{x@\pi'}\rangle\underline{\pi}\rangle\downarrow_E = (\underline{D}\langle\underline{\pi}\rangle\downarrow_E)\langle\langle\cdot\rangle\underline{x@\pi'}\downarrow_E\rangle$ is a right context because it is the composition of right contexts, given that $\underline{x@\pi'}\downarrow_E$ is a fireball by Lemma 11.2.

   (d) $s' = (D, \lambda x.\overline{t}, (y@\epsilon) : \pi, E) \rightsquigarrow_{\beta_1} (D, \overline{t}\{x\leftarrow y\}, \pi, E) = s$. By *i.h.* $R_{s'} = \underline{D}\langle\underline{(y@\epsilon) : \pi}\rangle\downarrow_E = \underline{D}\langle\langle\langle\cdot\rangle y\rangle\underline{\pi}\rangle\downarrow_E = (\underline{D}\langle\underline{\pi}\rangle\downarrow_E)\langle\langle\cdot\rangle y\downarrow_E\rangle$ is a right context, which implies that $R_s = \underline{D}\langle\underline{\pi}\rangle\downarrow_E$ is one such context as well.

   (e) $s' = (D, \lambda x.\overline{t}, \phi : \pi, E) \rightsquigarrow_{\beta_2} (D, \overline{t}, \pi, [x\leftarrow\phi]E) = s$ with $\phi \neq y@\pi'$. By *i.h.*, $R_{s'} = \underline{D}\langle\underline{\phi : \pi}\rangle\downarrow_E = \underline{D}\langle\langle\langle\cdot\rangle\underline{\phi}\rangle\underline{\pi}\rangle\downarrow_E = (\underline{D}\langle\underline{\pi}\rangle\downarrow_E)\langle\langle\cdot\rangle\underline{\phi}\downarrow_E\rangle$ is a right context, which implies that $\underline{D}\langle\underline{\pi}\rangle\downarrow_E$ is one such context as well. Now, $R_s = \underline{D}\langle\underline{\pi}\rangle\downarrow_{[x\leftarrow\phi]E} = \underline{D}\langle\underline{\pi}\rangle\downarrow_E$ because by Lemma 11.1b x may occur only in $\overline{t}$, and so the substitution $\downarrow_{[x\leftarrow\phi]E}$ acts on every code in D and π exactly as $\downarrow_E$.

   (f) $s' = (D, x, \phi : \pi, E) \rightsquigarrow_{s} (D, (\lambda y.\overline{u})^{\alpha}, \phi : \pi, E) = s$ where $E = E_1[x\leftarrow\lambda y.\overline{u}@\epsilon]E_2$. The invariant follows from the *i.h.* because $R_{s'} = R_s$, as the only component that changes is the code. □

Lemma 12 *(Fast GLAMOUr β-projection and overhead transparency).*[15] *Let s be a reachable state of a Fast GLAMOUr execution.*

1. Overhead Transparency*: if $s \rightsquigarrow_{s,c_{1,2,3}} s'$ then $\underline{s} = \underline{s'}$;*
2. β-Projection*: if $s \rightsquigarrow_{\beta_{1,2}} s'$ then $\underline{s} \to_{\mathtt{r}\beta_f} \underline{s'}$.*

Proof. Transitions:

1. $s = (D, \overline{t}\overline{u}, \pi, E) \rightsquigarrow_{c_1} (D : \overline{t}\Diamond\pi, \overline{u}, \epsilon, E) = s'$. Then

$$\underline{s} = \underline{D}\langle\langle\overline{t}\overline{u}\rangle\underline{\pi}\rangle\downarrow_E = \underline{D : \overline{t}\Diamond\pi}\langle\overline{u}\rangle\downarrow_E = \underline{D : \overline{t}\Diamond\pi}\langle\langle\overline{u}\rangle\epsilon\rangle\downarrow_E = \underline{s'}.$$

2. $s = (D : \overline{t}\Diamond\pi, \lambda x.\overline{u}, \epsilon, E) \rightsquigarrow_{c_2} (D, \overline{t}, (\lambda x.\overline{u}@\epsilon) : \pi, E) = s'$. Then

$$\underline{s} = \underline{D : \overline{t}\Diamond\pi}\langle\langle\lambda x.\overline{u}\rangle\underline{\epsilon}\rangle\downarrow_E = \underline{D}\langle\langle\overline{t}\langle\langle\lambda x.\overline{u}\rangle\underline{\epsilon}\rangle\rangle\underline{\pi}\rangle\downarrow_E = \underline{D}\langle\langle\overline{t}\rangle\underline{(\lambda x.\overline{u}@\epsilon) : \pi}\rangle\downarrow_E = \underline{s'}.$$

3. $s = (D : \overline{t}\Diamond\pi, x, \pi', E) \rightsquigarrow_{c_3} (D, \overline{t}, (x@\pi') : \pi, E) = s'$ with $E(x) = \bot$ or $E(x) = y@\pi''$ or $(E(x) = \lambda y.\overline{u}@\epsilon$ and $\pi' = \epsilon)$. Then

$$\underline{s} = \underline{D : \overline{t}\Diamond\pi}\langle\langle x\rangle\underline{\pi'}\rangle\downarrow_E = \underline{D}\langle\langle\overline{t}\langle\langle x\rangle\underline{\pi'}\rangle\rangle\underline{\pi}\rangle\downarrow_E = \underline{D}\langle\langle\overline{t}\rangle\underline{(x@\pi') : \pi}\rangle\downarrow_E = \underline{s'}.$$

4. $s = (D, \lambda x.\overline{t}, (y@\epsilon) : \pi, E) \rightsquigarrow_{\beta_1} (D, \overline{t}\{x\leftarrow y\}, \pi, E) = s'$. Then

$$\underline{s} = \underline{D}\langle\langle\lambda x.\overline{t}\rangle\underline{(y@\epsilon) : \pi}\rangle\downarrow_E = \underline{D}\langle\langle(\lambda x.\overline{t})y\rangle\underline{\pi}\rangle\downarrow_E \to_{\mathtt{r}\beta_f} \underline{D}\langle\langle\overline{t}\{x\leftarrow y\}\rangle\underline{\pi}\rangle\downarrow_E = \underline{s'}$$

   where the rewriting step takes place because $\underline{D}\langle\underline{\pi}\rangle\downarrow_E$ is a right context by Lemma 11.3.

5. $s = (D, \lambda x.\overline{t}, \phi : \pi, E) \rightsquigarrow_{\beta_2} (D, \overline{t}, \pi, [x\leftarrow\phi]E) = s'$ with $\phi \neq y@\epsilon$. Then

$$\underline{s} = \underline{D}\langle\langle\lambda x.\overline{t}\rangle\underline{\phi : \pi}\rangle\downarrow_E = \underline{D}\langle\langle(\lambda x.\overline{t})\underline{\phi}\rangle\underline{\pi}\rangle\downarrow_E \to_{\mathtt{r}\beta_f} \underline{D}\langle\langle\overline{t}\{x\leftarrow\underline{\phi}\}\rangle\underline{\pi}\rangle\downarrow_E = \underline{D}\langle\langle\overline{t}\rangle\underline{\pi}\rangle\{x\leftarrow\underline{\phi}\}\downarrow_E = \underline{D}\langle\langle\overline{t}\rangle\underline{\pi}\rangle\downarrow_{[x\leftarrow\phi]E} = \underline{s'}$$

   where the $\mathtt{r}\beta_f$-step takes place because $\underline{\phi}$ is a fireball by the fireball item invariant (Lemma 11.2), and $\underline{D}\langle\underline{\pi}\rangle\downarrow_E$ is a right context by the contextual decoding invariant (Lemma 11.3). Moreover, the meta-level substitution $\{x\leftarrow\underline{\phi}\}$ can be extruded (in the equality after $\to_{\mathtt{r}\beta_f}$) without renaming x, as x does not occur in D or π by the name invariant (Lemma 11.1b).

6. $s = (D, x, \phi{:}\pi, E) \rightsquigarrow_{s} (D, (\lambda y.\overline{u})^{\alpha}, \phi{:}\pi, E) = s'$ where $E = E_1[x\leftarrow\lambda y.\overline{u}@\epsilon]E_2$. Then, by Lemma 11.1a,

$$\underline{s} = \underline{D}\langle\langle x\rangle\underline{\phi : \pi}\rangle\downarrow_E = \underline{D}\downarrow_E\langle\langle x\downarrow_E\rangle\underline{\phi : \pi}\downarrow_E\rangle = \underline{D}\downarrow_E\langle\langle\lambda y.\overline{u}\downarrow_E\rangle\underline{\phi : \pi}\downarrow_E\rangle = \underline{D}\langle\langle\lambda y.\overline{u}\rangle\underline{\phi : \pi}\rangle\downarrow_E = \underline{s'}. \quad \square$$

Lemma 13 *(Fast GLAMOUr halt).*[16] *Let s be a reachable final state of a Fast GLAMOUr execution. Then $\underline{s}$ is a fireball, i.e. it is β_f-normal.*

[15] Stated at p. 19.
[16] Stated at p. 19.

Proof. An immediate inspection of the transitions shows that in a final state the code cannot be an application and the dump is necessarily empty. In fact, final states have one of the following three shapes:

1. *Top-level unapplied abstraction, i.e.* $s = (\epsilon, \lambda x.\bar{t}, \epsilon, E)$. Then $\underline{s} = (\lambda x.\bar{t})\downarrow_E = \lambda x.(\bar{t}\downarrow_E)$, which is an abstraction and so a fireball.
2. *Top-level free variable, i.e.* $s = (\epsilon, x, \epsilon, E)$ (note that, differently from the Easy GLAMOUr, it might be $E(x) \neq \bot$). We claim that $\underline{s} = x\downarrow_E$ is a fireball. Suppose not: then, the only possibility to have $x\downarrow_E$ different from a fireball would be that an item ϕ in E is such that $\underline{\phi}\downarrow_E$ is not a fireball, but this is impossible by the fireball item invariant (Lemma 11.2).
3. *Top-level compound inert term, i.e.* $s = (\epsilon, x, \phi{:}\pi, E)$ with $E(x) \neq \lambda y.\bar{t}@\epsilon$. Subcases:
   (a) $E(x) = \bot$. Then $\underline{s} = (\langle x\rangle\underline{\pi})\downarrow_E = \langle x\downarrow_E\rangle(\underline{\pi}\downarrow_E) = \langle x\rangle(\underline{\pi}\downarrow_E)$. By the fireball item invariant (Lemma 11.2), $\underline{\phi}\downarrow_E$ is a fireball for every item ϕ in π, so $\langle x\rangle(\underline{\pi}\downarrow_E)$ is an inert term and hence a fireball.
   (b) $E(x) = y@\pi'$. Then $\underline{s} = (\langle x\rangle\underline{\pi})\downarrow_E = \langle x\downarrow_E\rangle(\underline{\pi}\downarrow_E) = \langle \underline{y@\pi'}\downarrow_E\rangle(\underline{\pi}\downarrow_E)$. By the fireball item invariant (Lemma 11.2), $\underline{y@\pi'}\downarrow_E$ is inert and $\underline{\phi}\downarrow_E$ is a fireball for every item ϕ in π, so $\langle \underline{y@\pi'}\downarrow_E\rangle(\underline{\pi}\downarrow_E)$ is inert and hence a fireball. □

References

[1] S. Abramsky, C.L. Ong, Full abstraction in the lazy lambda calculus, Inf. Comput. 105 (2) (1993) 159–267, https://doi.org/10.1006/inco.1993.1044.
[2] B. Accattoli, The useful MAM, a reasonable implementation of the strong λ-calculus, in: Logic, Language, Information, and Computation – 23rd International Workshop, WoLLIC 2016, in: Lect. Notes Comput. Sci., vol. 9803, 2016, pp. 1–21, https://doi.org/10.1007/978-3-662-52921-8_1.
[3] B. Accattoli, The complexity of abstract machines, in: 3rd International Workshop on Rewriting Techniques for Program Transformations and Evaluation, WPTE 2016, invited paper, in: EPTCS, vol. 235, 2017, pp. 1–15, https://doi.org/10.4204/EPTCS.235.1.
[4] B. Accattoli, (In)efficiency and reasonable cost models, Electron. Notes Theor. Comput. Sci. 338 (2018) 23–43, https://doi.org/10.1016/j.entcs.2018.10.003.
[5] B. Accattoli, B. Barras, Environments and the complexity of abstract machines, in: 19th International Symposium on Principles and Practice of Declarative Programming, PPDP 2017, 2017, pp. 4–16, https://doi.org/10.1145/3131851.3131855.
[6] B. Accattoli, U. Dal Lago, Beta reduction is invariant, indeed, in: 23rd Conference on Computer Science Logic and 29th Symposium on Logic in Computer Science, CSL-LICS '14, 2014, pp. 8:1–8:10, https://doi.org/10.1145/2603088.2603105.
[7] B. Accattoli, G. Guerrieri, Open call-by-value, in: Programming Languages and Systems – 14th Asian Symposium, APLAS 2016, in: Lect. Notes Comput. Sci., vol. 10017, 2016, pp. 206–226, https://doi.org/10.1007/978-3-319-47958-3_12.
[8] B. Accattoli, G. Guerrieri, Implementing open call-by-value, in: Fundamentals of Software Engineering – 7th International Conference, FSEN 2017, in: Lect. Notes Comput. Sci., vol. 10522, 2017, pp. 1–19, https://doi.org/10.1007/978-3-319-68972-2_1.
[9] B. Accattoli, G. Guerrieri, Types of Fireballs, in: Programming Languages and Systems – 16th Asian Symposium, APLAS 2018, in: Lect. Notes Comput. Sci., vol. 11275, 2018, pp. 45–66, https://doi.org/10.1007/978-3-030-02768-1_3.
[10] B. Accattoli, C. Sacerdoti Coen, On the relative usefulness of fireballs, in: 30th Symposium on Logic in Computer Science, LICS 2015, 2015, pp. 141–155, https://doi.org/10.1109/LICS.2015.23.
[11] B. Accattoli, C. Sacerdoti Coen, On the value of variables, Inf. Comput. 255 (2017) 224–242, https://doi.org/10.1016/j.ic.2017.01.003.
[12] B. Accattoli, P. Barenbaum, D. Mazza, Distilling abstract machines, in: 19th International Conference on Functional Programming, ICFP 2014, 2014, pp. 363–376, https://doi.org/10.1145/2628136.2628154.
[13] B. Accattoli, P. Barenbaum, D. Mazza, A strong distillery, in: Programming Languages and Systems – 13th Asian Symposium, APLAS 2015, in: Lect. Notes Comput. Sci., vol. 9458, 2015, pp. 231–250, https://doi.org/10.1007/978-3-319-26529-2_13.
[14] B. Accattoli, A. Condoluci, G. Guerrieri, C. Sacerdoti Coen, Crumbling abstract machines, in: 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, 2019, pp. 4:1–4:15, https://doi.org/10.1145/3354166.3354169.
[15] Z.M. Ariola, A. Bohannon, A. Sabry, Sequent calculi and abstract machines, ACM Trans. Program. Lang. Syst. 31 (4) (2009) 13:1–13:48, https://doi.org/10.1145/1516507.1516508.
[16] H.P. Barendregt, The Lambda Calculus – Its Syntax and Semantics, Stud. Logic Found. Math., vol. 103, North-Holland, 1984.
[17] G.E. Blelloch, J. Greiner, A provable time and space efficient implementation of NESL, in: International Conference on Functional Programming, ICFP 1996, 1996, pp. 213–225, https://doi.org/10.1145/232627.232650.
[18] T. Coquand, G.P. Huet, The calculus of constructions, Inf. Comput. 76 (2/3) (1988) 95–120, https://doi.org/10.1016/0890-5401(88)90005-3.
[19] P. Crégut, An abstract machine for lambda-terms normalization, in: LISP and Functional Programming, LFP 1990, 1990, pp. 333–340, https://doi.org/10.1145/91556.91681.
[20] U. Dal Lago, S. Martini, On constructor rewrite systems and the lambda-calculus, in: Automata, Languages and Programming – 36th International Colloquium, Part II, ICALP 2009, in: Lect. Notes Comput. Sci., vol. 5556, 2009, pp. 163–174, https://doi.org/10.1007/978-3-642-02930-1_14.
[21] U. Dal Lago, S. Martini, Derivational complexity is an invariant cost model, in: Foundational and Practical Aspects of Resource Analysis – 1st International Workshop, FOPARA 2009, in: Lect. Notes Comput. Sci., vol. 6324, 2009, pp. 100–113, https://doi.org/10.1007/978-3-642-15331-0_7.
[22] O. Danvy, I. Zerny, A synthetic operational account of call-by-need evaluation, in: 15th International Symposium on Principles and Practice of Declarative Programming, PPDP '13, 2013, pp. 97–108, https://doi.org/10.1145/2505879.2505898.
[23] M. Fernández, N. Siafakas, New developments in environment machines, Electron. Notes Theor. Comput. Sci. 237 (2009) 57–73, https://doi.org/10.1016/j.entcs.2009.03.035.
[24] D.P. Friedman, A. Ghuloum, J.G. Siek, O.L. Winebarger, Improving the lazy Krivine machine, High.-Order Symb. Comput. 20 (3) (2007) 271–293, https://doi.org/10.1007/s10990-007-9014-0.
[25] Á. García-Pérez, P. Nogueira, J.J. Moreno-Navarro, Deriving the full-reducing Krivine machine from the small-step operational semantics of normal order, in: 15th International Symposium on Principles and Practice of Declarative Programming, PPDP '13, 2013, pp. 85–96, https://doi.org/10.1145/2505879.2505887.
[26] J.-Y. Girard, P. Taylor, Y. Lafont, Proofs and Types, Camb. Tracts Theor. Comput. Sci., vol. 7, Cambridge University Press, 1989.
[27] B. Grégoire, X. Leroy, A compiled implementation of strong reduction, in: 7th International Conference on Functional Programming, ICFP '02, 2002, pp. 235–246, https://doi.org/10.1145/581478.581501.
[28] R. Harper, F. Honsell, G.D. Plotkin, A framework for defining logics, in: Symposium on Logic in Computer Science, LICS '87, 1987, pp. 194–204.
[29] G. Nadathur, D. Miller, An overview of lambda-PROLOG, in: Logic Programming – 5th International Conference and Symposium, ICLP/SLP 1988, 1988, pp. 810–827.
[30] L. Paolini, S. Ronchi Della Rocca, Call-by-value solvability, RAIRO Theor. Inform. Appl. 33 (6) (1999) 507–534, https://doi.org/10.1051/ita:1999130.

[31] G.D. Plotkin, Call-by-name, call-by-value and the lambda-calculus, Theor. Comput. Sci. 1 (2) (1975) 125–159, https://doi.org/10.1016/0304-3975(75)90017-1.
[32] S. Ronchi Della Rocca, L. Paolini, The parametric lambda calculus – a metamodel for computation, Texts in Theoretical Computer Science. An EATCS Series, Springer, 2004, https://doi.org/10.1007/978-3-662-10394-4.
[33] D. Sands, J. Gustavsson, A. Moran, Lambda calculi and linear speedups, in: The Essence of Computation, Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones, in: Lect. Notes Comput. Sci., vol. 2566, 2002, pp. 60–84, https://doi.org/10.1007/3-540-36377-7_4.
[34] P. Sestoft, Deriving a lazy abstract machine, J. Funct. Program. 7 (3) (1997) 231–264.
[35] Terese, Term Rewriting Systems, Camb. Tracts Theor. Comput. Sci., vol. 55, Cambridge University Press, 2003.
[36] M. Wand, On the correctness of the Krivine machine, High.-Order Symb. Comput. 20 (3) (2007) 231–235, https://doi.org/10.1007/s10990-007-9019-8.